

微型计算机基础知识教育丛书（新版）

C 语言基础教程 （修订版）

吕凤翥 编著



北京大学出版社

微型计算机基础知识教育丛书(新版)

C 语言基础教程

(修订版)

吕凤翥 编著

北京大学出版社
北 京

内 容 简 介

本书作者总结了十多年来专心从事 C 语言教学的经验,全面系统地讲解了 C 语言的基本语法和基础的编程方法。本书包含有词法和语法规则,常量、变量,运算符和表达式,语句,函数和存储类,预处理功能,指针,结构,联合和枚举,以及文件操作等内容。本书在讲述上具有突出重点、详述难点、揭示疑点的特点。有近 150 个不同类型的例题,每章都有丰富的思考题和作业题。本书语言通俗、概念准确、抓住读者心理,回答读者问题,适于自学。

本书可作为高等院校本科和大专学生的教材和教员的教学参考书,也可作为成人教育、自学考试教材和参考书。

图书在版编目(CIP)数据

C 语言基础教程/吕凤翥编著. —北京:北京大学出版社,1998.2
(微机计算机基础知识教育丛书)
ISBN 7-301-03668-X

I. C... II. 吕... III. C 语言-程序设计-基本知识 IV. TP312

书 名: C 语言基础教程(修订版)

著作责任者: 吕凤翥

责任编辑: 杨锡林

标准书号: ISBN 7-301-03668-X/TP · 391

出 版 者: 北京大学出版社

地 址: 北京市海淀区中关村北京大学校内 100871

电 话: 出版部 62752015 发行部 62559712 编辑部 62752032

排 印 者: 北京市兴盛达激光照排中心

发 行 者: 北京大学出版社

经 销 者: 新华书店

787×1092 16 开本 18.25 印张 450 千字

1998 年 3 月第一版 1998 年 3 月第一次印刷

定 价: 26.00 元

序 言

从第一台电子计算机问世到今天,几近半个世纪,人类从生产到生活发生了巨大的变化,电脑已悄然闯入社会生活的各个领域。过去说:没有电将寸步难行;现在要说:没有计算机就没有现代化。

计算机科学是信息科学的一个重要组成部分。21 世纪将以信息技术为主导,使整个社会的经济活动方式与社会的就业结构产生非常大的变化。体力劳动的比重将逐渐减少,掌握信息技术的脑力劳动者的比例将不断增大。电子函件、电子新闻、电子图书等新的科技将逐步取代纸笔和印刷机,新的计算机文化将迅速发展。

著名的计算机科学家 G·伏赛斯曾预言:电脑将是继自然语言、数学之后而成为第三位的,对人的一生都有大用处的“通用智力工具”。现在,实践已经证明了电脑已经成为各行各业的基本工具。许多部门已经把具备电脑的应用知识与技能作为录用或考核工作人员的一个重要条件。综合国力的竞争说到底还是掌握高科技人才的竞争。怎样将计算机科学知识迅速而有效地普及到全社会,也就成了一件具有紧迫感的新任务。

近年来为适应社会的需求,各类职业教育学校有了较快的发展。在这些学校里的学生理所当然地要接受计算机教育。但是,目前的状况是,适用于这些学校的教材却非常之少。因此,尽快写出这种教材供同学们选用,是我们编写“微型计算机基础知识教育丛书”的初衷。从教学目标出发,这套丛书将重点讲述基本概念和基本方法,以理论联系实际思路介绍一些具体的实际操作技术;在写作手法上,力求通俗而不肤浅,深入而不玄奥,贯彻循序渐进的原则;在每一应知应会的知识点上,着力讲深讲透;书中附有必要的思考题和上机练习题,引导读者既动脑又动手,学深,学活,学以致用。

随着电脑应用的普及,蒙在电脑上的一块神秘的面纱已经被揭落。许多学过电脑的人都感到,入门不难,深造也是办得到的。只要功夫深,电脑不会不听命。

中国计算机学会普及委员会主任
清华大学计算机科学与技术系教授

吴文虎
1997.3.25

前 言

本书是在《实用C语言基础教程》和《C语言入门》(这两本书都由北京大学出版社出版)的基础上,根据广大读者的需要,总结了十多年来从事C语言教学的经验编写而成的。

本书在内容上和编排上都作了较大的调整,较明显地体现出突出重点、详述难点、揭示疑点的特点。本书经修改后,内容上更全面和系统,它包含了C语言所有的基础知识和基本技能;讲解上更加深入细致,先讲清概念,指出方法和规则,然后提醒在实际应用中的注意事项,最后通过列举例题加深理解和学会应用,对每个例题都做了详尽的分析,并指出编程的技巧和方法。本书的例题增加了,全书近150个不同类型的例题,每个例题都具有一定的代表性,有的是训练语法的,有的是练习编程的,从每个例题中都会得到收获;作业题的内容更加丰富了,有判断、有填空、还有分析程序结果和编程,既可训练对基础知识的理解,又可学会分析问题、解决问题的方法,提高C语言的编程能力。

作者根据十多年来从事C语言教学的经验,较好地抓住读者学习过程中的问题,并在书中予以解答。总之,修订后的本书更适合作为教材和自学指导书。

本书共分十章。由浅入深系统地全面地讲述了C语言的语法知识和编程技术。前三章讲述了C语言的发展、特点和应用,C语言的词法及其规则,C语言编程特点;还讲述了C语言的基础知识:常量、变量、类型转换、运算符和表达式等内容,这里强调了运算符的种类、功能、使用方法、优先级和结合性等,这是一个重点和难点。第四至六章讲述了C语言中主要的语法内容:语句和函数,同时讲解了存储类和预处理功能;这部分重点讲解函数的定义和说明,函数参数和返回值以及函数的调用,突出两种不同形式的传值调用;这部分难点是变量的存储类和各类标识符的作用域。第七章指针是C语言的最大特点,指针也是学习C语言的最大难点,本书用较大篇幅详尽地讲述了指针的概念和指针数组和函数方面的具体应用;强调在实际应用中理解指针概念和使用指针的方法;本章通过大量例题展示出指针的广泛应用,从中掌握指针的本质及运算规则,这对学会使用指针有极大帮助。第八、九两章,讲述了C语言的三种构造的数据类型:结构、联合和枚举;这里,结构是重点;通过讲述结构的概念和应用,学会结构的使用方法,并看到结构在解决实际问题中的重要性。最后一章讲述了文件的操作,它包括C语言文件的特点,标准文件的读写函数的操作方法、一般文件的打开和关闭操作、读写操作和定位操作等。

本书每章后面都有复习本章内容的思考题和检验对本章内容理解和掌握程度的作业题,作业题中包含有训练概念的填空和判断题,练习操作方法的分析结果题和培养编程能力的编程题。作业题中的分析程序结果题和编程题可以上机验证,从而得知你的分析是否正确和你编写的程序是否符合要求。这些题目不仅种类较多,而且类型也较全,重复性很小,不仅适用读者练习,也可用于教员选取作为学生的作业题,或稍加变化可作为考试题。

当你读完本书后,可能对C语言产生了浓厚的兴趣,还想继续使用C语言进行程序设计的方法,你可以继续读本人编写的《C语言应用教程》(该书已由北京大学出版社出版),它会帮助你学会C语言的编程方法,同时会看到C语言的运用范围可真广啊!

本书的所有程序(例题和作业题中程序)都在 Turbo C 语言编译系统下调试过,并备有程序软盘。

由于时间较紧,难免会有错误存在,请读者指正。

谢谢喜欢我编著的计算机书籍的读者。

作 者

1997 年 10 月于北大燕北园

目 录

第一章 C语言概述	(1)
1.1 C语言的由来与发展	(1)
1.1.1 C语言的由来	(1)
1.1.2 C语言的发展	(1)
1.2 C语言的特点和应用	(2)
1.2.1 C语言的特点	(2)
1.2.2 C语言的应用	(5)
1.3 C语言的词法及其规则	(6)
1.3.1 字符集	(6)
1.3.2 单词及词法规则	(6)
1.4 C语言常用的输入输出函数	(9)
1.4.1 常用的输入函数	(9)
1.4.2 常用的输出函数	(10)
1.5 C语言程序实例及其实现	(12)
1.5.1 C语言程序实例.....	(12)
1.5.2 C语言程序书写格式.....	(14)
1.5.3 C语言程序实现	(15)
练习题	(17)
作业题	(17)
第二章 常量、变量和类型转换	(19)
2.1 常量	(19)
2.1.1 数字常量	(19)
2.1.2 字符常量和字符串常量	(20)
2.1.3 符号常量	(22)
2.2 变量	(23)
2.2.1 变量的名字	(23)
2.2.2 变量的类型	(24)
2.2.3 变量的值	(25)
2.3 数组	(27)
2.3.1 数组的定义	(27)
2.3.2 数组的赋值	(28)
2.3.3 字符数组	(31)
2.4 类型转换	(34)
2.4.1 自动转换	(34)
2.4.2 强制转换	(35)
练习题	(35)

作业题	(36)
第三章 运算符和表达式	(38)
3.1 常用运算符的功能	(38)
3.1.1 算术运算符	(38)
3.1.2 增 1 减 1 运算符	(39)
3.1.3 关系运算符	(40)
3.1.4 逻辑运算符	(40)
3.1.5 位操作运算符	(41)
3.1.6 赋值运算符	(42)
3.1.7 其他运算符	(43)
3.2 运算符的优先级和结合性	(45)
3.2.1 运算符的优先级	(45)
3.2.2 运算符的结合性	(45)
3.3 表达式	(47)
3.3.1 表达式和表达式的种类	(47)
3.3.2 表达式的值和类型	(56)
3.3.3 表达式求值中值得注意的两个问题	(57)
练习题	(60)
作业题	(61)
第四章 语句	(63)
4.1 表达式语句和空语句	(63)
4.1.1 表达式语句	(63)
4.1.2 空语句	(64)
4.2 复合语句和分程序	(64)
4.2.1 复合语句	(64)
4.2.2 分程序	(65)
4.3 分支语句	(65)
4.3.1 条件语句	(65)
4.3.2 开关语句	(70)
4.4 循环语句	(76)
4.4.1 while 循环语句	(76)
4.4.2 do-while 循环语句	(77)
4.4.3 for 循环语句	(79)
4.5 转向语句	(85)
4.5.1 goto 语句	(85)
4.5.2 break 语句	(86)
4.5.3 continue 语句	(87)
4.5.4 return 语句	(88)
练习题	(89)
作业题	(89)
第五章 函数和存储类	(95)

5.1	函数的定义和说明	(95)
5.1.1	函数的定义	(95)
5.1.2	函数的说明	(97)
5.2	函数的参数和返回值	(97)
5.2.1	函数的参数	(97)
5.2.2	函数的返回值	(99)
5.3	函数的调用	(100)
5.3.1	传值调用的特点	(100)
5.3.2	传址调用的特点	(101)
5.3.3	数组名作参数的函数调用	(103)
5.3.4	函数的嵌套调用	(105)
5.3.5	函数的递归调用	(106)
5.4	作用域规则	(110)
5.4.1	标识符的作用域规则	(110)
5.4.2	重新定义变量的作用域规定	(111)
5.5	存储类	(112)
5.5.1	变量的存储类	(112)
5.5.2	函数的存储类	(118)
	练习题	(122)
	作业题	(122)
第六章	预处理功能和类型定义	(129)
6.1	预处理功能概述	(129)
6.2	宏定义	(129)
6.2.1	简单宏定义	(130)
6.2.2	带参数的宏定义	(132)
6.2.3	宏定义的应用	(135)
6.3	文件包含	(136)
6.3.1	文件包含命令的格式和功能	(136)
6.3.2	使用文件包含命令时应注意事项	(137)
6.4	条件编译	(139)
6.4.1	条件编译的常用命令格式	(139)
6.4.2	条件编译命令的应用	(140)
6.5	类型定义	(142)
6.5.1	类型定义的含意和类型定义语句	(142)
6.5.2	类型定义的应用	(144)
	练习题	(144)
	作业题	(145)
第七章	指针	(150)
7.1	指针的概念	(150)
7.1.1	什么是指针	(150)
7.1.2	指针的表示	(151)

7.1.3	指针的赋值	(153)
7.1.4	指针所指向变量的值	(155)
7.2	指针的运算	(157)
7.2.1	指针的赋值运算	(157)
7.2.2	指针加减整数的运算	(157)
7.2.3	两个指针相减的运算	(157)
7.2.4	两个指针比较的运算	(158)
7.2.5	指针运算与地址运算的区别	(159)
7.3	指针与数组	(160)
7.3.1	数组名是一个常量指针	(160)
7.3.2	数组元素的指针表示	(161)
7.3.3	字符数组、字符指针和字符串处理函数	(168)
7.3.4	指向数组的指针和指针数组	(173)
7.4	指针与函数	(179)
7.4.1	指针作函数参数	(180)
7.4.2	指针函数和指向函数的指针	(185)
	练习题	(190)
	作业题	(190)
第八章	结构	(198)
8.1	结构的概念	(198)
8.1.1	结构和结构变量的定义	(198)
8.1.2	结构变量成员的表示	(200)
8.1.3	结构变量的赋值	(201)
8.1.4	结构变量的运算	(203)
8.2	结构与数组	(204)
8.2.1	数组与结构成员	(204)
8.2.2	结构数组	(205)
8.3	结构与函数	(209)
8.3.1	结构变量与指向结构变量的指针作函数参数	(209)
8.3.2	结构变量和指向结构变量的指针作函数返回值	(212)
8.4	链表	(213)
8.4.1	链表的概念	(213)
8.4.2	链表的操作	(214)
8.5	位段	(224)
8.5.1	位段的概念	(224)
8.5.2	使用位段时应注意的事项	(226)
	练习题	(227)
	作业题	(227)
第九章	联合和枚举	(233)
9.1	联合的概念	(233)
9.1.1	联合变量的定义和赋值	(233)

9.1.2 联合与结构的区别	(234)
9.2 联合的应用	(236)
9.3 枚举的概念.....	(239)
9.3.1 枚举变量的定义和赋值	(239)
9.3.2 使用枚举变量时应注意的事项	(240)
9.4 枚举的应用	(241)
练习题.....	(243)
作业题.....	(243)
第十章 文件和读写函数	(246)
10.1 C 语言中文件的概念	(246)
10.1.1 文件和文件指针	(246)
10.1.2 标准文件和一般文件	(247)
10.1.3 高级读写函数和低级读写函数	(248)
10.2 标准文件的读写操作	(248)
10.2.1 标准文件读写函数介绍	(248)
10.2.2 标准文件读写函数应用	(251)
10.3 一般文件的操作.....	(255)
10.3.1 打开文件函数和关闭文件函数	(255)
10.3.2 一般文件读写函数及其使用	(256)
10.3.3 文件定位函数及其使用	(263)
10.4 介绍常用的其他函数	(266)
10.4.1 动态存储分配函数	(266)
10.4.2 系统调用函数	(267)
10.4.3 字符函数	(268)
10.4.4 常用数学函数	(269)
练习题.....	(269)
作业题.....	(270)
附录	(276)
附录 1 ASCII 编码表	(276)
附录 2 ctype.h 文件中所包含的字符函数	(278)
附录 3 math.h 文件中所包含的数学函数	(279)

第一章 C 语言概述

本章介绍 C 语言的由来和发展、C 语言的特点和应用、C 语言程序的书写格式和实现,使读者对 C 语言程序结构有一个初步的了解,并对 C 语言建立一个整体概念。本章还介绍 C 语言程序中常用的输入输出函数以及 C 语言的词法和词法规则,为后面各章的学习打下一个基础。

1.1 C 语言的由来与发展

1.1.1 C 语言的由来

C 语言诞生于 1972 年,由美国电话电报公司(AT&T)贝尔实验室的 D. M. Ritchie 设计,并首先在一台使用 UNIX 操作系统的 DEC PDP-11 计算机上实现。

C 语言是在一种称为 B 语言的基础上,克服了 B 语言依赖于机器又无数据类型等局限性开发的语言。在 1970 年,美国贝尔实验室的 K. Thompson 以 BCPL 语言为基础,设计出一种既简单又接近于硬件的 B 语言,并用它写成了第一个 UNIX 操作系统,在 PDP-7 计算机上实现的。B 语言是取了 BCPL 语言的第一个字母。而 BCPL 语言(Basic Combined Programming Language)是 1967 年英国剑桥大学的 M. Richards 基于一种 CPL 语言(Combined Programming Language)提出的一种改进的语言。而 CPL 语言又是于 1963 年英国剑桥大学根据 ALGOL 60 推出的一种接近硬件的语言。由此可见,C 语言的根源可以追溯到 ALGOL 60,它的演变过程如下所示:

ALGOL60(1960 年)→CPL(1963 年)→BCPL(1967 年)→B(1970 年)→C(1972 年)

1.1.2 C 语言的发展

C 语言是在人们设想寻找一种既具有一般高级语言的特征,又具有低级语言特点的语言的情况下应运而生的,它具有人们的这种期望,集中了高级语言和低级语言的优点。最初的 C 语言就是为了描述和实现 UNIX 操作系统而产生的一种工具语言。1973 年,贝尔实验室的 K. Thompson 和 D. M. Ritchie 两人合作使用 C 语言修改了 UNIX 操作系统,即 UNIX 第 5 版本。原来的 UNIX 操作系统是用汇编语言写的,改写后 UNIX 操作系统中 90%以上使用了 C 语言。从此,C 语言的命运与 UNIX 操作系统便有着密切的联系,随着 UNIX 操作系统的发展和推广,C 语言也在被广泛的使用和发展。

C 语言出世以后,在应用中不断的改进。在 1975 年 UNIX 第 6 版本公布以后,C 语言开始引起人们的注意,它的优点逐步被人们所认识。1977 年出现了与具体机器无关的 C 语言编译文本,推动了 UNIX 操作系统在各种机器上的迅速地实现。随着 UNIX 的日益广泛的使用,C 语言也得到了迅速的推广。1978 年以后,C 语言先后被移植到大、中、小和微型机上,它很快成为世界上应用最广泛的计算机语言之一。

1978 年又推出 UNIX 第 8 版本,以该版本中的 C 编译程序为基础,B. W. Kernighan 和

D. M. Ritchie 合作(被称为 K&R)出版了《The C Programming Language》(C 程序设计语言)一书,被称为标准 C。1983 年,ANSI(美国国家标准化协会)对 C 语言的各种版本进行了扩充,推出了新的标准,被称为 ANSI C,它比原来的标准 C 有了改进和扩充。1987 年,ANSI 又公布了 87 ANSI C 新版本。目前流行的各种 C 语言编译系统的版本大多数都以此为基础,但各有其不同。当前微机上使用的 C 语言编译系统多为 Microsoft C, Turbo C, Borland C 和 Quick C 等,它们略有差异,按标准 C 书写的程序,基本上都可运行。读者要了解不同版本的编译系统的特点和区别可参阅有关的操作说明书。

1.2 C 语言的特点和应用

1.2.1 C 语言的特点

C 语言是一种开发比较晚的高级语言,它吸取了早期高级语言的优点,克服了某些不足,形成了它独有的特性。C 语言的特点概括起来有如下几点。

1. C 语言是一种结构化的程序设计语言

结构化程序设计的优点是便于分工合作,便于调试、维护和扩充。这种程序设计方法是将一个程序分成若干个模块,每个模块完成一个功能,由一个总控模块来控制 and 协调各个模块来实现总的功能;因此,这种程序设计方法又称为模块化程序设计方法。在 C 语言中,函数是构成 C 语言程序的最小模块。实际上,C 语言的程序是由一个或者多个文件组成的,每个文件又是由一个或多个函数组成的。因此,一个程序是由一个或多个函数组成的,其中,须有且仅有一个主函数,主函数的名字规定为 `main()`。这样,组成一个程序的若干个文件中,仅有一个是主文件,只有主文件中才含有 `main()` 函数。另外,函数是由若干条语句组成的,语句是由若干个单词组成的,单词是由若干个字符组成的。字符是构成程序的最小单元。C 语言程序的构成如下所示:

程序→文件→函数→语句→单词→字符。

C 语言是结构化程序设计语言,它具有构成三种基本结构模式的语句,这种结构化程序设计应具有三种基本结构模式如下:

● 顺序结构模式

它将由若干条顺序执行的语句构成,这是程序设计的最基本形式。

● 分支结构模式

C 语言中具有条件语句和开关语句,它将会构成各种需要的分支结构模式。

● 循环结构模式

C 语言中提供了三种循环语句(`for` 循环, `while` 循环和 `do-while` 循环),用它们可以构成各种循环结构模式。

在讲述 C 语言是一种结构化程序设计语言的同时,还必须指出它与其他结构化程序设计语言(如 PASCAL 语言)相比较,还有一些不完全之处,因此,C 语言是一种不完全的结构化程序设计语言。其表现如下:

第一,完全的结构化程序设计语言不允许使用 `goto` 语句,因为 `goto` 语句会破坏结构化。但是,`goto` 语句在某些时候会使得程序简练,因此,在 C 语言中原则上允许使用 `goto` 语句,为了减少它对结构化的影响,采取了限制使用的办法,即规定 `goto` 语句的转向范围只能在一个

函数体内,不得使用 goto 语句从一个函数体内转向到另一个函数体中,这种限制性地使用 goto 语句会给编程带来一些方便,又不会影响模块之间的结构化。但是,建议在 C 语言的程序中尽量少使用 goto 语句。

第二,完全的结构化程序设计语言要求一个模块只有一个入口和一个出口,这样便于结构化的管理。但是,C 语言程序中允许函数使用多个返回语句(return 语句),即允许函数有多个出口,返回到调用函数。这样做也是为了编程中的方便。在 C 语言程序中,可以使用条件语句来返回不同的函数值。

由此可见,C 语言虽然是一个不够严格的结构化程序设计语言,但它是一个使用起来十分灵活的高级语言。

2. C 语言十分简练

C 语言是一种非常简练的语言。用 C 语言编写的程序十分简洁。C 语言的简洁性表现如下:

(1) C 语言中类型说明符采用缩写形式,例如,整型可用 int,而不用 integer;字符型用 char 而不用 character;长整型可用 long 等等。

(2) C 语言中关键字较少,只有 32 个。有些关键字用简单的符号代替,例如,条件语句中的 if 体的定界符采用花括号({}),如果是一条语句规定不用定界符。又例如,循环语句中循环体也是如此。

(3) 运算符丰富。不仅数量多,而且功能强,例如,三目运算符(?:)具有条件语句的功能。三目运算符使用方法如下:

```
d1? d2:d3
```

其中,d1,d2,d3 是不同的表达式。其功能是先计算表达式 d1 的值,如果 d1 的值是非零,则上述三目运算符组成的条件表达式的值为 d2 表达式的值;否则(即 d1 表达式的值为零)条件表达式的值为 d3 的值。因此,它将相当如下表示的 if 语句:

```
if (d1)
    c=d2;
else
    c=d3;
```

其中,c 用来存放上述条件表达式的值。

使用功能强的运算符可以使得程序简洁。

(4) 预处理功能将简化程序书写内容。C 语言中提供了一种预处理功能,它包含宏定义和文件包含等,其中宏定义有一种简化书写的功能,它可将一个复杂的格式用一个简单格式来定义,例如

```
#define PI 3.14159265
```

表示定义 PI 为 3.14159265,在程序中只需用 PI 表示,在编译前系统将用 3.14159265 来替代 PI。这里,简化是一个特点,还有其他好处后面会介绍。还有文件包含也将起到简化书写的目的,它将许多文件都将使用的一些语句放在某个指定的文件(一般用头文件,即.h 文件)中,那么在某个文件中需要这些语句时,不必重写,只要将原来写好的指定文件包含进来就可以了,其方法十分简单,例如

```
#include "myfile.h"
```

其中,myfile.h 是事先写好的需要包含其内容的文件。

3. C 语言功能很强

C 语言的功能性强表现在它既具有高级语言的功能又具有低级语言的功能。数值运算和非数值运算的功能 C 语言都具有,并且在处理非数值数据时更加方便和灵活。此外,C 语言还具备一些低级语言的功能,例如,寄存器运算功能,二进制位运算功能和内存地址运算功能等。这些低级语言的功能是一般高级语言所没有的。由于 C 语言具有上述这些功能,因此,它的应用十分广泛,它不仅像其他高级语言一样编写一些应用程序,而且还可以像汇编语言一样编写一些系统程序。而在实际上,许多系统软件,例如关系数据库管理系统,绘图软件系统等,都是用 C 语言编写的。另外,使用 C 语言编写一些接口程序也十分方便。

C 语言还提供了丰富的数据类型,除了基本的数据类型之外,还提供了构造的数据类型,例如,数组、结构、联合和枚举。使用这些数据类型可以很方便地实现各种复杂的数据结构(例如,链表、栈、树等)的操作。

4. C 语言的可移植性好

高级语言的可移植性都比汇编语言好。在诸多的高级语言中,C 语言的移植性更为突出。这是因为该语言编译系统较小,另外预处理功能对移植也带来一些方便,因此,C 语言本身只需稍加修改便可用于各种型号的机器上和各类操作系统中,用 C 语言编写的程序也很方便地用于不同系统中,这也是 C 语言得以广泛应用的原因之一。

在了解和掌握 C 语言上述的特点的同时,还应该知道 C 语言所存在的不足,这些不足往往是由于突出某个特点而带来的。在学习 C 语言中,了解这些不足是很重要的,它可以避免出现一些莫名其妙的错误。下面列举四个方面的不足。

(1) 运算符多,难用难记。C 语言中有 40 多个运算符,又分 15 种优先级和两类结合性,这无疑对数据的运算和处理带来了方便。但是,诸多的运算符和不同的优先级势必会带来难记忆,难使用的不足。例如,有些功能不同的运算符,却使用相同的符号,如“*”,它作为单目运算符表示取内容,它作为双目运算符表示两个操作数相乘,它与斜线符“/”连在一起(/* 或 */)表示注释符,另外,它用在说明语句中,用来表示它右边变量为指针等等。还有运算符的 15 种优先级不要记混了,否则会造成计算值的错误。

(2) C 语言中类型转换比较灵活,例如,int 型与 char 型可以自动转换。这些自动转换的规定带来一定的计算方便,在 C 语言中允许一个字符与一个整数进行加减运算,例如

```
'a'+2
```

是合法的,该表达式的值用字符型表示为 'c',用整型数表示为 99。为了转换上的方便,在许多情况下将不作类型检查,例如,在只作简单说明时的函数调用中,要求形参和实参类型一致,如果类型不一致也不判错仍可通过,有时会造成运算结果的错误,为了避免这类错误的发生,可以对函数进行原型说明,这时编译时将会对形参和实参的类型进行检查,增加其安全性。另外,也可使用强制类型运算符来限制其不必要的自动转换。

(3) C 语言中数组在动态赋值时不作越界检查。因此,在数组元素的个数少于实际赋值的项数时,编译系统不报越界错,而继续按数组元素所表示的地址进行赋值,这样容易造成数据的混乱。为了避免这种情况的发生,应尽量避免动态的越界赋值。对数组的静态赋值(即赋初值)不会发生越界赋值的现象,静态时出现越界赋值将报错。

(4) C 语言为了优化等原因允许不同的编译系统在表达式或参数表内重新安排求值顺序。这样对于一般的表达式值和参数表内各项参数值是没有影响的,例如,改变表达式中各操作数的计算顺序不会改变表达式的值。但是,对一些具有副作用的运算符(如,增 1 减 1 运算符和赋值运算符)来讲,不同的求值顺序将会造成表达式的不同的值。这一点在编写程序时应特别注意,避免出现那些可能有二义性的表达式和参数表,关于这方面详细讲解请见本书后面内容。

1.2.2 C 语言的应用

从前面对 C 语言的特点的分析中,不难看出 C 语言具有编程方便、语句简练、功能很强、移植性好等优点,它是编程者喜欢使用的一种结构化程序设计语言。

C 语言已被广泛地应用于系统软件和应用软件的开发中。在下述的几个方面应用得更为广泛。

1. 数据库管理和应用程序方面

C 语言的非数值处理功能很强,因此它被广泛地应用于数据库管理系统和应用软件。大多数的关系数据库管理系统,如 dBASE, FoxBASE, ORACLE 等,都是由 C 语言编写的。各种不同部门的应用软件也大都是用 C 语言开发的,C 语言在开发数据库应用软件方面应用很广,深受开发者的欢迎。

2. 图形图像系统的应用程序方面

C 语言在图形图像的开发中也有着广泛的市场。很多图形图像系统,如 AutoCAD 通用图形系统等,就是使用 C 语言开发的,并且在这些图形系统中可以直接使用 C 语言编程,实现某些功能。C 语言编译系统带有许多绘图功能的函数,利用这些函数开发图形应用软件十分方便。所开发的应用程序常用 C 语言编写接口界面,这样既方便又灵活,效果很好。这是因为该语言提供有图形处理功能,便于实现图形图像的各种操作。因此,C 语言在图形图像的应用方面很好地发挥了它的作用。

3. 编写与设备的接口程序方面

C 语言不仅在建立友好界面方面有着广泛应用,如下拉式菜单、弹出菜单、多窗口技术等;而且在编写与设备的接口程序方面也有着广泛应用。这是因为 C 语言不仅具有高级语言的特性还具有低级语言的功能,因此,在编写接口程序方面十分方便,有时它与汇编语言一起使用,会显示出更高的效率。

4. 数据结构方面

由于 C 语言提供了十分丰富的数据类型,不仅有基本数据类型还有构造的数据类型,如数组、结构和联合等,把它们用于较复杂的数据结构(例如,链表、队列、栈、树等)中显得十分方便,这方面已有许多成熟的例程供选择使用。

5. 排序和检索方面

排序和检索是数据处理中最常遇到并较为复杂的问题。使用 C 语言来编写排序和检索各种算法的程序既方便又简洁。特别是有些排序算法采用了递归方法进行编程,更显得清晰明了。因此,人们喜欢使用 C 语言来编写这方面的程序。

上述列举了五个方面的应用,但绝不是说 C 语言的应用仅限如此,而是说在这几个方面目前使用得更多些。C 语言可以说在各个领域中都可以使用,并且都会有较好的效果。所以,C

语言是当前被用于编程的最广泛的语言之一。

另外,C语言是一种结构化程序设计语言,在编写大型程序中也很方便,特别是该语言又提供了预处理功能,其中文件包含在多人同时开发一个大程序时将带来减少重复和提高效率等好处,因此,越来越多的人喜欢用C语言来开发大型程序。

1.3 C语言的词法及其规则

1.3.1 字符集

字符是构成C语言程序的最小单元,若干字符组合成单词。下面给出C语言中使用的合法字符。

1. 字母和数字

小写字母:a b c d e f g h i j k l m n o p q r s t u v w x y z

大写字母:A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

数字:0 1 2 3 4 5 6 7 8 9

2. 标点和特殊字符

字符	名称	字符	名称	字符	名称
,	逗号	{	左花括号	#	# 字号
.	点	}	右花括号	%	百分号
;	分号	<	小于号	&	和号
:	冒号	>	大于号	^	脱字符
'	单撇号	!	惊叹号	*	星号
"	双撇号		竖线	-	减号
(左圆括号	/	斜线	=	等号
)	右圆括号	\	反斜线	+	加号
[左方括号	~	求反号		
]	右方括号	_	下划线		

3. 空白符和空字符

空白符包含空格符、回车符、换行符、水平制表符等。

空字符是指ASCII码值为零的那个字符。该字符在C语言中有特殊用处,用它来作为字符串的结束符。

以上字符的集合称为C语言的合法字符集。

1.3.2 单词及词法规则

单词是由若干个有序的字符组成的,单词的集合称为词汇。C语言的单词有如下几种:标识符、关键字、运算符、分隔符、常量、字符串和注释符。

下面对上述7种单词的词法规则逐一详述,有些单词,例如,关键字、运算符、常量和字符串等,后面的章节中还会详细讲述。

1. 标识符

标识符是由字母、下划线和数字组成的字符序列,要求第一个字母必须是字母或下划线。标识符是用来给 C 语言程序中所使用的变量、函数、语句标号、类型定义等起名字的。C 语言本身对标识符所用字符个数不作限制,但是在具体使用中,有些计算机只识别前面 8 个字符,而其他字符不作识别。另外,对大写小写字母是区分的。例如,a 和 A 分别表示不同的变量。

在使用标识符起名字时,要注意尽量有意义并便于阅读。一般变量名或函数名多以小写字母开始或全部用小写字母,例如,a2, ab, creat_list()等。有人喜欢将表示某种含义的多个英文单词组成的名字中用下划线作为单词间的连接等,如建立链表函数起名为 creat_list()。也有人喜欢将多个英文单词连在一起写而不用下划线分隔,为了表示不同单词而将除第一个单词之外的其他单词的第一个字母大写,如删除链表某结点的函数起名为 deleteNode()。

下列的标识符是合法的:

x100, int_cnt, _xy, aB, sum, lotus_1_2_3, studentName 等。

下面的标识符是非法的:

26d, #mon, ab.c, \$xy, M. John, $x \geq y$, $m * n$, 4B 等。

读者要学会写出合法的标识符,也要能识别出非法的标识符。

在给变量、函数等起名字时最好能做到“见名知意”,即从标识符的字符集中可知道该变量或函数的含意。例如,year, month, day, name, age, sex 等,不难从英文单词中了解变量的含意。

2. 关键字

关键字是一种具有特定含意的标识符。关键字又称保留字。因为这些标识符是系统已经定义过的,不能再定义了,需要加以保留。使用者不能用关键字作为所定义的标识符,因此,读者要知道系统已经使用了哪些保留字。下面将它们分类说明如下(关于这些关键字的含意及用法本书后面章节会有详细讲解)。

(1) 标识类型的关键字

int, char, long, float, double, short, unsigned, struct, union, enum, auto, extern, static, register, typedef, void。

(2) 标识控制流的关键字

goto, return, break, continue, if, else, default, do, while, switch, case。

(3) 标识预处理功能的关键字

define, include, undef, ifdef, ifndef, endif, line 等。

(4) 其他关键字

sizeof, asm, fortran, ada, pascal 等。

上述的标识符都是系统已有定义的保留字,读者不得再重新定义。

3. 运算符

运算符是用来表示某种运算操作的一种符号,有的运算符用一个字符组成,也有的运算符由多个字符组成。有的运算符只要求有一个操作数,这种运算符叫单目运算符;有的运算符要求有两个操作数,称为双目运算符;还有要求有三个操作数的运算符,称为三目运算符。C 语言中运算符种类繁多,优先级复杂,还有结合性等问题,本书第三章中专门讲述运算符问题。

4. 分隔符

分隔符是用来分隔多个变量、数据项、表达式等的符号。C 语言中常用的分隔符有逗号, 空白符, 分号和冒号。下面介绍这些分隔符的使用方法。

(1) 逗号作为分隔符用来分隔多个变量和函数参数。例如, 在说明语句中, 同种类型的多个变量可用逗号将其变量分隔开:

```
int a, b, c, d;
```

这表示变量 a, b, c 和 d 都被定义为 int 型变量。又例如, 在函数定义或调用时, 用逗号将函数的多个形参或实参进行分隔, 在下列函数的定义中,

```
add(a, b, c)
int a, b, c;
{
    :
```

这表示函数 add 有三个形参 a, b 和 c, 用逗号进行分隔。

(2) 空白符一般常用来作为多个单词间的分隔符, 也可以作为输入数据时自然输入项的缺省分隔符。C 语言中, 语句是由单词组成的, 单词之间不能连写在一起, 中间要用空白符(常用空格符)作分隔符, 例如,

```
int a, b, c;
```

这里, 在 int 与 a 之间便是用空格符分开的, 因为 int 是一个单词, a 又是一个单词, 这两个单词不能写在一起, 一定要用空白符分隔开。又例如, 使用标准格式输入函数输入数据时, 并且控制串中又没有指定匹配符, 则从键盘上输入的数据项之间需用空白符作为分隔符。关于标准格式输入函数 scanf(), 下面将会详细介绍。

(3) 分号有时也可作为分隔符使用, 它主要用在 for 循环语句中 for 后面, 圆括号内的三个表达式之间用分号分隔, 这一点将在语句一章中详细讲解。

(4) 冒号有时也可作为分隔符使用, 它主要用于语句标号与语句之间, 用冒号分开; 也可用在 switch 语句中, case 关键字与其后的语句之间用冒号分开。这些使用在本书后面的内容中都会讲到。

5. 常量

常量是一种在程序中其值保持不变的量。C 语言中常量分为数字常量和字符常量两类。

数字常量又分为整型常量和浮点型常量。

字符常量和字符串常量是两种不同的常量。

C 语言中, 常量通常用符号常量来表示, 符号常量也是一种标识符。

有关常量的类型、常量的使用方法以及符号常量的定义等详细情况后面章节会专门描述。

6. 字符串

字符串是一种由双引号(" ")括起来的一串字符组成的常量。字符串实际上被存放在一个字符数组中。

下面是合法的字符串:

"xyz", "This is a string", "1257", "Wang ping", " ", "a\b\"等。

其中, " "表示一个空串, 即没有字符的串。"a\b\"表示由 a"b"组成的字符串, 当双引

号本身作为字符处理时,前面要加一个反斜线,以便与定界符的双引号加以区别。

在C语言中,要注意字符常量与字符串常量的区别。它们在表示上、用法上和存放上都是不同。字符常量是用单引号(' ')括起来的单个字符,例如,'a'是一个字符常量,字符串常量是用双引号(" ")括起来的一个或多个字符组成;字符常量在一定条件下可与整数进行加法或减法运算,字符串常量有与字符常量不同的运算;字符常量存放在内存中占一个字节的空间,字符串常量存放在内存中占有的字节个数是字符个数加1,因为每个字符串存放在内存中都有一个结束符'\0'。例如,字符'a'占1个字节,而"a"却占用2个字节,其中'a'占一个,结束符'\0'占一个。

7. 注释符

注释符是用来标识注释或提示信息的。程序中的注释信息不被编译也不被执行,其作用是增加程序的可读性。C语言的注释符是以/*开头并以*/结束,在/*和*/之间的信息为注释信息,一般起到说明或备忘的作用。

注释符可以出现在程序中任意行的位置,既可在程序头,也可在程序尾,还可以在程序中的任意行。注释符可出现在一条语句的前面,也可出现在一条语句的后边,甚至还可出现在一条语句的中间。注释信息可占一行,也可以占多行。注释符在有些编译系统中允许嵌套,即在注释信息中还可以包含注释符。例如,

```
/* Compute /* Squares, Circle */ */
```

这是注释符嵌套的形式,这种形式有的编译系统是不允许的,有的编译系统经过设置后是允许的。

另外,采用注释符的形式在调试程序中常常有用,可以将一些暂时不参与编译和运行的语句用注释符加以注释,一旦去掉注释符后,仍可参与编译或运行。

1.4 C语言常用的输入输出函数

这里先介绍几个常用的输入输出函数,因为这些函数将在程序中经常出现,关于其余的输入输出函数将在“文件和读写函数”一章中讲述。

1.4.1 常用的输入函数

常用的输入函数是指从键盘上接收数据的函数,它们是getchar(), gets()和scanf()三个函数。

1. 获得一个字符的函数 getchar()

该函数的功能是从键盘上获取一个字符,它是带缓冲区和回显的,所谓带缓冲区是指该函数不是当一个字符键入后立即被接收,而是将键入的字符先放在内存缓冲区中,当若干个字符键入完后,再从缓冲区中按先后顺序获得字符。所谓带回显是指键入一个字符后在显示器屏幕上显示出所键入的字符。该函数的格式如下所示:

```
int getchar()
```

该函数没有参数,它的返回值是一个int型数,即所接收的字符的ASCII码值。

2. 获得一个字符串的函数 gets()

该函数的功能是从键盘上获取所键入的字符串。该函数的正常返回值是一个字符型指针,

即读取到的字符串的首地址,出错时返回 NULL(NULL 被定义为 0)。该函数的格式如下所示

```
char * gets(s)
char * s;
```

其中,* 作为说明符表示指针,而 char * 表示 char 型指针。具体指针的详细讲解在本书“指针”一章中。输入的字符串以 '\n'(换行符)为结束。

3. 标准格式输入函数 scanf()

标准格式输入函数是指从标准输入设备键盘上读取数据;并且按所指定的格式将读取的数据赋给相应的变量。该函数的格式如下:

```
int scanf("控制串",<参数表>)
```

该函数的参数由两部分组成,其中一部分是由双引号括起来被称为控制串,另一部分是参数表。〈控制串〉中包含格式符和一般字符。格式符是用来说明对应的输入项的格式的。格式符的标识符是百分号(%),它后面跟的字母表示格式的格式说明符。scanf()函数的格式说明符如下所示:

- d——十进制整数
- x——十六进制整数
- o——八进制整数
- u——无符号十进制数
- f——小数表示的浮点数
- e——指数表示的浮点数
- c——单个字符
- s——字符串

控制串中的一般字符表示匹配符,另外在%和格式说明符之间还可加修饰符,这些内容将在“文件和读写函数”一章中讲解。

〈参数表〉是由一个或多个参数构成,多个参数使用时用逗号分隔。每个参数用地址值表示。要求参数的个数和类型与控制串中格式符的个数和类型相一致,即要求其个数相等,类型相同。

该函数具有一个整型数的返回值,该返回值表示该函数参数表中成功获得数据的参数的个数。

三种输入函数的例子,将会在后面的程序中看到。

1.4.2 常用的输出函数

常用的输出函数是指将输出结果显示在屏幕上的函数,它们是 putchar(), puts()和 printf()三个函数。

1. 输出一个字符的函数 putchar()

该函数的功能是将所指定的一个字符输出到屏幕上,即将该字符显示在屏幕上。该函数的格式如下:

```
int putchar(c)
int c;
```

其中,c 是该函数的参数。该函数将 c 所表示的字符显示在屏幕上。c 可以是一个字符常

量,也可以是一个字符型变量,还可以是一个表达式。正常情况下,该函数返回输出字符的代码值。出错时,返回 EOF。

2. 输出一个字符串的函数 puts()

该函数的功能是将所指定的字符串显示在屏幕上。其格式如下:

```
int puts(s)
char * s;
```

其中,s 是该函数的参数,该参数指出要输出显示的字符串,它可以是一个字符串常量,也可以是一个字符型数组,或是一个指向字符串的指针。该函数正常时返回零。

3. 标准格式输出函数 printf()

该函数是将指定的表达式的值按指定的格式输出到屏幕上,即显示在屏幕上。该函数的格式如下:

```
int printf("控制串",<参数表>)
```

该函数的参数可分两个部分:一部分是<控制串>,用双引号括起;另一部分是<参数表>,中间用逗号分隔。<控制串>中包含有格式符和一般字符。格式符是百分号作为标识符,其后用一个字母表示输出格式,该字母称为格式说明符。该函数的格式说明符如下所示:

- d——十进制整数
- o——八进制整数
- x——十六进制整数
- u——无符号整数
- c——单个字符
- s——字符串
- f——浮点数(小数型)
- e——浮点数(指数型)
- g——e 和 f 中较短的一种

在格式标识符(%)与格式说明符之间可以使用修饰符,用来限制输出数据的宽度和对齐方式。常用的修饰符如下:

数字. 数字——小数点前面的数字用来表示输出数据的最小域宽,所谓最小域宽是指当输出数据的实际宽度小于最小域宽度时,按最小域宽输出数据,一般用空格符补到最小域宽;当输出数据的实际宽度大于最小域宽时,则按实际宽度输出数据。可见最小域宽是用来指出输出数据的最小宽度。小数点后面的数字用来表示输出数据的精度,对浮点数来讲表示小数点后的位数;对字符串来讲表示输出字符串的最大个数,并将超过的部分截掉;对整数来讲表示输出的最大位数,超过的部分被截去(很少使用)。

l——用于格式说明符 d, o, x 前边表示长整数,用于 e, f, g 前面表示双精度浮点数。

-——负号用来表示数据在域宽中左对齐。如果不用负号,则表示右对齐。

还有一些其他修饰符将在“文件和读写函数”一章中讲述。

控制串中的一般字符照样输出,即将一般字符显示在屏幕上。对于一般的不可打印字符用转义序列表示,而可打印字符直接用字符符号表示。所谓转义序列是用来表示字符的一种方法,即用该字符的 ASCII 码值来表示,具体格式如下:

\ddd

在反斜线后面跟三位八进制数字来表示字符的 ASCII 码值,也可用十六进制数字表示字符的 ASCII 码值,其格式为\xhh。例如,表示字符<ESC>可用\033 或\x1b 表示。为了表示方便,C 语言中将一些常用的控制代码的转义序列用\<字母>形式表示,如表 1-1 所示。

表 1-1

转义序列	含 义	转义序列	含 义
\n	换行	\r	回车
\t	水平制表	\v	垂直制表
\b	退格	\f	走纸换页
\a	鸣铃	\\	反斜线符
\'	单引号符	\0	ASCII 码为零的字符

(参数表)是由 0 个或多个参数组成,多个参数用逗号分隔。0 个参数表示没有参数表,即参数表前的逗号可省略,这里只有控制串部分,常用这种格式在屏幕上输出提示信息。参数表中每个参数是一个表达式。要求参数表中参数的个数和类型与控制串中的格式符的个数和类型相一致,即要求其个数相等,类型相同。这里提醒注意的是当其个数不等类型不同时,一般情况下,不报错误信息,而会造成输出结果上的不一致。在使用该函数时应特别注意这一点。该函数的返回值是一个整数,它通常表示输出显示的数据的总宽度。

关于上述三个输出函数的例子,将会在后面的程序中看到。

1.5 C 语言程序实例及其实现

1.5.1、C 语言程序实例

在讲解 C 语言语法之前,先列举几个 C 语言程序的实例,读者可以从中了解到 C 语言程序的书写格式,对 C 语言程序有个初步了解。

[例 1.1] 编写一个程序输出如下字符串:

This is a program.

程序内容如下:

```
main()
{
    printf("This is a program. \n");
}
```

这是一个很简单的程序。该程序只有一个文件,并只有一个函数 main(),它是一个主函数,并没有参数。该函数的函数体是用一对花括号({})括起来的。函数体内只有一个语句。注意,一条语句的最后要有一个分号(;),这是 C 语言程序的一个特点。该语句是前面介绍过的标准格式输出函数 printf(),在该函数中只有用双引号括起来的控制串部分,没有任何参数,因此,该函数将双引号内的字符串输出显示在屏幕上,在字符串中除了最后有一个 '\n' 字符外,都是一般可打印字符,而 '\n' 是用转义序列表示的换行符。

执行该程序后,则在屏幕上显示如下信息:

This is a program.

光标在字符串的下一行开始处。

[例 1.2] 编写一个程序,求出给定的两个数的和。

程序内容如下:

```
main()
{
    int a,b,sum;
    printf("Input a and b: ");
    scanf("%d%d",&a,&b);
    sum=add(a,b);
    printf("sum=%d+%d=%d\n",a,b,sum);
}
add(x,y)
int x,y;
{
    return(x+y);
}
```

该程序是由两个函数组成的,这两个函数分别是主函数 main() 和被调用函数 add()。函数 add() 的功能是将两个参数之和返回给调用函数,即将其返回值(x+y 的值)赋给变量 sum。在 main() 中,首先说明了三个 int 型变量:a、b 和 sum。接着,使用 printf() 函数在屏幕上显示出如下的提示信息。

Input a and b:

表示提醒读者要从键盘上输入两个 int 型数分别给变量 a 和 b。然后,通过执行 scanf() 函数,将键盘上输入的两个数分别赋给了变量 a 和 b,即使 a 和 b 从键盘上获得了数值。这里使用了标准格式输入函数 scanf(), 该函数的控制串中仅有两个格式符,而参数表中对应有两个参数,&a 和 &b 分别表示了变量 a 和变量 b 的地址值,这里的 & 是一个取变量地址的运算符(后面再讲),可以看到 scanf() 函数中,格式符与参数的个数是相等的,类型也是相同的。

接着,程序中使用了调用函数语句,将被调用函数的返回值赋给变量 sum。调用函数时使用了两个参数 a 和 b,即 add(a, b)。调用时将实参 a 和 b 的值分别传送给 add 函数的两个形参 x 和 y。可见,实参和形参是个数相等,对应的类型相同的。被调用函数的两个形参 x 和 y 获得由实参 a 和 b 传送的值以后,通过一个返回语句 return,将表达式 x+y 的值返回给调用函数,并将控制权也返回给主函数,于是主函数中的变量 sum 便获得了返回来的值,即 x+y 的值。再接着,执行 printf() 函数,按其规定的格式在屏幕上显示出结果。printf() 函数中的控制串有三个格式符,对应着参数表中的三个表达式,它们对应的类型是相同。

执行该程序时,屏幕上显示出如下提示信息:

Input a and b:

这里,如果从键盘上键入如下信息:

18 52

则在屏幕上将显示出如下结果:

sum=18+52=70

通过上述的两个例子,可以对 C 语言程序有个初步了解。分析上述程序可以看到:

(1) C 语言程序是由函数构成的。例 1.1 是一个只有一个函数构成的简单程序,该函数一定要是主函数 main()。例 1.2 是一个由两个函数构成的程序,其中一个是主函数 main(),另一个是被调用函数 add()。可见 C 语言程序可由若干个函数构成,其中必须有一个且只能有一个是 main(),其余的都是被调用的函数,这便是 C 语言程序的一大特点。

(2) 函数是 C 语言程序的基本单位。函数是由两部分组成的:一部分称为函数头,它是函数的说明部分,包含函数类型、函数名、一对圆括号、函数参数(形参)名和参数的说明;另一部分称为函数体,函数体是由一对花括号起来的由若干条语句组成的,这对花括号标识了函数体的范围。有关函数定义的详情在后面的“函数和存储类”一章还有介绍。

(3) 程序执行时总是从 main() 函数开始的,main() 可放在程序的任何位置,程序中所有函数都是并行的,函数之间不存在包含(即嵌套)关系,只存在调用关系。

1.5.2 C 语言程序书写格式

C 语言具有语句简洁的特点,C 语言程序的可读性比较差。因此,为了增强 C 语言的可读性,正确的书写格式就显得十分重要。同样一个程序采用不同的书写方法,尽管都可以得到相同的结果,有的书写方法可读性强,有的书写方法可读性差。读者不妨分析下列程序的输出结果。

[例 1.3] 分析下列程序的输出结果:

程序内容如下:

```
multiply(x,  
y)  
int  
x,y;{return(x*y);  
}main(){  
int a,  
b;a=5;  
b=6;printf(  
"%d\n",  
multiply  
(a,b)  
);  
}
```

读者一定会发现这个程序不容易读懂,主要是书写上没有按照习惯的格式书写。C 语言书写要求比较自由,一般只要一个单词不得分开写,单词之间用空白符分隔,而空白符包含空格符、水平制表符和换行符等。因此,例 1.3 中的程序书写上并没有词法错误。执行该程序后,会在屏幕上显示出如下结果:

30

如果将该程序重新书写一遍,如下所示。

[例 1.4] 将例 1.3 程序重新书写如下:

```
multiply(x,y)  
int x,y;  
{
```

```

        return(x * y);
    }
    main()
    {
        int a,b;
        a=5;
        b=6;
        printf("%d\n",multiply(a,b));
    }

```

这种书写格式比较好,使得程序比较好读。该程序由两个函数组成:main()函数是主函数,multiply()函数是被调用函数。执行该程序后,先执行 main()函数,主函数中定义了 a 和 b 是 int 型变量,并给 a 和 b 分别赋了值。在 printf()函数中,有一个参数,该参数是调用 multiply()函数,通过将实参 a 和 b 的值传送给对应的形参 x 和 y,在被调用函数中通过返回语句将表达式 x * y 的值返回给调用函数,即将返回值按 %d 的格式显示在屏幕上,其结果为 30。

可见,书写格式与可读性有很大关系。下面将给出书写 C 语言程序的常用格式以及应注意的事项。

(1) C 语言程序一般是一行写一个语句,也可以一行写几个语句,还可以一个语句写几行。一般情况下,一个语句写在多行上不用续行符。但是,有的编译系统有时需要用续行符进行续行,即在续行的前一行末尾加上反斜线(\)这个续行符。语句不需加行号,只有在 goto 语句转向到的语句前需加语句标号。

(2) 每个语句(包括说明语句和执行语句)的末尾必须有一个分号(;)。分号是语句的组成部分,不可缺少,即使程序中最后一个语句也要加分号。要注意:一个语句没有结束,在换行前不要加分号。

(3) 花括号的对齐方式常用的有三种格式(K&R, Allman, whitesmiths),本书采用了 Allman 格式,该格式规定每个花括号都单独成行,并且左花括号和闭花括号都与使用它们的语句对齐,而花括号中的内容向右缩进两个字符。

在说明语句中,使用花括号作为初始值表时和在枚举类型定义中用花括号作为枚举符表的定界时,花括号不必独占一行。例如:

```

int a []={1,2,3,4};
enum color{black, blue, red, yellow};

```

(4) 在书写程序中可使用 /* ... */ 对任何部分注释,以增加程序的可读性。

1.5.3 C 语言程序实现

学习 C 语言离不开编写和运行 C 语言程序,在了解一些 C 语言的初步知识以后,就应该上机练习编写和运行 C 语言的程序,通过上机实践来加深对 C 语言的认识和理解。

如何实现 C 语言程序呢?在不同的环境下实现的方法稍有差异。C 语言程序实现可归纳如下三步:

1. 编辑

编辑是用 C 语言写出源程序。其方法有两种:一种是使用编辑程序编写好 C 语言源程序,并以 .C 为后缀存入文件系统;另一种是使用 C 语言编译系统提供的编辑器来编写源程序,并

且存入文件系统。

2. 编译连接

编译连接是两个过程,有些编译系统常将它们连在一起,实际上是将源程序先进行编译,通过编译可发现源程序中的语法错误。如有错误,则系统将其“错误信息”显示在屏幕上,用户根据指出的错误信息,对源程序进行编辑修改,修改后再重新编译,直到编译无错为止。编译后生成机器指令程序,被称为目标程序。此目标程序名与相应的源程序同名,其后缀为.obj。编译过程完成后,便开始连接过程。所谓连接是将目标程序与库函数或其他程序连接成为可执行的目标程序,简称可执行程序。一般可执行程序名同源文件名,后缀为.exe。

3. 运行

当程序编译连接后,生成了可执行程序便可运行了。这里,还需补充一点,在连接过程中可能出现错误,这时必须根据“出错信息”所指示的错误进行修改后,再进行连接直到不出错为止,这样才会生成可执行文件。运行可执行文件,一般屏幕上显示出输出结果。

运行C语言程序的环境很多,编译系统也很多,不同环境其实现方法不同,但都包含了上面描述的三步。

下面以DOS系统下Turbo C 2.0版本的C语言编译系统为例,因为该系统流行较为广泛。

首先在Turbo C编译程序的目录下,在DOS命令提示符后键入下述命令:

tc ↵

屏幕顶部显示出如下所示的菜单行:

FILE EDIT RUN COMPILE PROJECT OPITONS DEBUG BREAK/WATCH

共有8个菜单项。通过移动←键或→键来选择菜单项,或者用鼠标选择。

当选取FILE项按回车后,屏幕上出现如图1.1所示的下拉式菜单窗口。

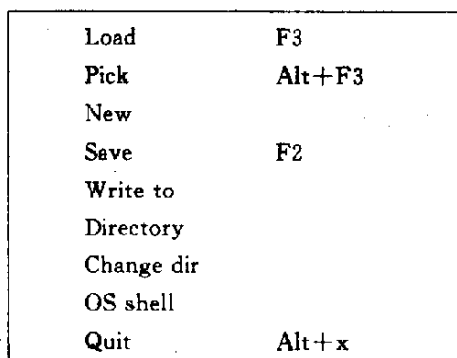


图 1.1

使用↑键或↓键选择子菜单项。选择“Load”项按回车键后,屏幕出现如图1.2所示的小窗口。要求键入要装入的文件名。当键入已有的文件名时,屏幕上将显示出该文件的内容。

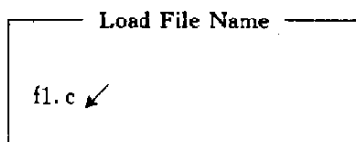


图 1.2

如果键入一个新文件名时,则可编辑一个新文件。选择“New”项按回车键后,可直接编辑新文件,编辑后选择“Write to”项按回车键后,在屏幕上出现的小窗口中键入文件名,再按回车键后便可存盘。

当源文件编辑完成,并已存盘,便可进行编译连接了。这时,按“F9”键,便对源程序进行编译,如果有错误,则按显示在屏幕上的错误信息对源文件进行修改,直到没有编译错为止,接着对编译后生成的.obj文件进行连接。在连接的过程中,如果有错误,屏幕上也显示“出错信息”,这时需在“Edit”状态下进行编辑修改,直到连接无错为止,这时生成可执行文件。

运行程序时,按“F10”键,屏幕顶部又出现主菜单行,选择“Run”项,按回车键后,则运行已编译连接好的可执行文件,执行后在屏幕上显示该程序应输出的结果。

如果发现输出结果不对,则还要重新修改源程序。这时,可通过按“F10”键,出现主菜单行,选择“Edit”项,便进入编辑状态,根据需要进行修改,修改完毕后,再度编译连接和运行,直到输出满意结果为止。

在编辑好一个源程序后,可以直接选择主菜单中“Run”项,这时对源程序先编译,无错后再连接,连接无错后生成可执行文件,接着便进行运行,将输出结果显示在屏幕上。

此外,Turbo C也具有命令行的编译连接方法。在C语言的源程序编辑好后,可在DOS系统的提示信息后面,键入如下信息:

```
tcc -c ff f1.c
```

便对源程序f1.c先进行编译,无错后再进行连接,连接无错后生成可执行文件,其名字为ff.exe,该可执行文件的名字是用户在上述命令行中选项-e后指出的,也可以指定其他名字。在编译和连接的过程中,如果出现错误信息,则需按其提示进行修改,修改后再进行编译连接,直到没错为止。

编译连接后生成了指定名字的可执行文件,接着,执行该程序,便可在DOS系统提示符后直接输入可执行文件名。例如,

```
ff
```

这时,ff.exe文件便被执行。该程序的输出结果将被显示在屏幕上。

练 习 题

1. 简述C语言的由来和发展,说明C语言与UNIX系统的关系。
2. C语言有哪些突出的特点?它又有哪些不足?应该如何看待这些不足?
3. C语言有哪些主要的应用?它为什么既用来开发系统软件又用来编写应用程序?
4. C语言的字符集包含哪些字符?
5. C语言中有哪些单词?各自有什么规定?
6. 标识符和关键字有什么区别?字符常量与字符串常量有何不同?
7. C语言中有哪些分隔符?它们各自的作用是什么?
8. 常用的输入函数有哪些?scanf()函数的作用是什么?如何使用该函数?
9. 常用的输出函数有哪些?printf()函数的作用是什么?如何使用该函数?
10. C语言程序在书写格式上有哪些特点?

作业题

1. 填空

(1) C 语言程序是由若干个()组成的,一个()又是由若干个()组成的,而()是组成 C 语言程序的基本单元。

(2) C 语言程序中函数是由若干条()组成的,每条()是由若干个()组成的,每个()是由若干个有序的()组成的。()是组成函数的最小元素。

(3) 为了从键盘上给整型变量 x, y 和 z 赋值,使用标准格式输入函数 scanf(),其格式如下:

```
scanf("___", ___, ___, ___);
```

(4) 为将一个浮点型变量 time 的值和一个字符串"computer"输出到屏幕上,并要求浮点数小数后取 2 位,可使用如下语句:

```
printf("___", ___, ___);
```

(5) 语句中各单词的分隔符是(),函数中多个参数之间的分隔符是(),语句标号与语句间的分隔符是(),说明多个变量时,多个变量之间的分隔符是()。

(6) 下列标识符中,()是错的。

A. xyMn B. a10 C. fl.c D. m-1

(7) 下列对 C 语言程序结构的描述中,()是错误的。

A. C 语言程序是由若干个函数构成的

B. 程序中的函数是由函数头和函数体两个部分组成的

C. C 语言程序可由多个文件组成,要求每个文件中只能有一个主函数 main()

D. C 语言程序中对主函数 main()放置的位置没有一定要求,可放在前面,也可放在后面,还可放在中间

(8) 下列对 C 语言程序的书写格式的描述中,()是对的。

A. C 语言程序中,每行只能写一个语句

B. C 语言程序的续行符是反斜线(\)

C. C 语言程序的书写中要求每行都要以分号(;)作结尾

D. 注释行必须放在程序头或程序尾

2. 分析下列程序的输出结果,并上机验证。

```
main()
{
    int x,y;
    x=18;
    y=16;
    y=x+y;
    printf("%d\n",y);
}
```

3. 练习编写一个程序,输入如下信息:

```
? ? ? ? ? ? ? ? ? ? ?
      MENU
* * * * *
```

4. 编写一个 C 语言程序,用来从两个数中选出最小的。

5. 上机运行本章的例 1.1,例 1.2,例 1.3。

第二章 常量、变量和类型转换

本章讲解 C 语言中基础内容：常量和变量。常量和变量是 C 语言程序中所不可缺少的基本量。常量和变量是两种不同性质的量，本章讲解它们的定义和表示。数组是一种变量的类型，在 C 语言中它属于构造的数据类型，在实际问题中应用较多，本章讲解数组的定义和赋值。本章还要介绍类型转换的规则，在后面的表达式中要使用这些规则。本章内容比较简单，它为后面的学习打下基础。

2.1 常 量

这里讲解常量的种类、表示及其用法。常量分为数字常量和字符常量。在 C 语言中，常量多是通过符号常量来表示的。

2.1.1 数字常量

数字常量包括整型常量和浮点型常量，浮点型常量又称实数。

1. 整型常量

C 语言中整型常量有三种不同的表示形式：十进制、八进制和十六进制。

(1) 十进制。这是一种常用的表示形式，它将直接给出数字，即在数字前不加任何前缀。例如，12,259,703 等为十进制表示。

(2) 八进制。表示八进制数字时，要加前缀 0，即在数字前面加 0。例如，017,0532,0416 等为八进制表示。其中，017 换成十进制数为 15。

(3) 十六进制。表示十六进制数字时，要加前缀 0x 或 0X，即在数字前面加 0x 或 0X。例如，0x17,0xae5,0X4f 等为十六进制表示。其中，0x17 换成十进制数为 23。

整型数又分为长整数、短整数和无符号整数。长整型数在表示上与其他整型数的区别是加后缀 L 或 l，所谓后缀是指在数字后面加写的字母。例如，12345L,0x76abfL 等。无符号数的后缀是 U 或 u。例如，7643U,04216u 等。

[例 2.1] 不同进制数的表示及相互转换。

```
main()
{
    printf(" %d\t%x\t%o\n",29,29,29);
    printf(" %d\t%x\t%o\n",025,025,025);
    printf(" %d\t%x\t%o\n",0x1a,0x1a,0x1a);
    printf(" %u\t%ld\n",47675u,74261L);
}
```

请读者分析该程序执行后的输出结果。

2. 浮点型常量

浮点型常量又称为实型常量。它有两种表示形式：十进制小数形式和指数形式。

(1) 十进制小数形式。它是由数字和小数点组成的(必须有小数点)。一般数值不是很大或很小的数采用小数形式表示,这种形式方便易读。例如,0.567,567.0,.567,567.,0.0等都是合法的小数表示形式。

(2) 指数表示形式。指数表示法又称科学记数法。该表示形式中,须有字母e或E,且在该字母之前必须有数字,在该字母之后的指数必须为整数。对于过大或过小的数值采用这种表示方法。这种表示方法简明清晰。具体格式如下所示:

〈整数部分〉.〈小数部分〉e〈指数部分〉

字母e左边部分可以是〈整数部分〉.〈小数部分〉,也可以只有〈整数部分〉不含小数点,或者只有〈小数部分〉前面含有小数点,〈指数部分〉是整数,可以是正的,也可以是负的。例如,下列浮点数都是正确的:

1234e3, 1234E-2, 12.34e-3, .1234E5, 0e0 等。

而下面的浮点数表示是错误的:

e2, 3.5e1.5, .e5, e 等。

其出错原因是:e2在字母e前面没有数字;3.5e1.5是字母e后面指数为小数;.e3在字母e前面只有小数点(.)而没有数字;e是在字母e前无数字。

另外,关于负的浮点数的表示,例如,-5.0,-3.2e5等,一般应归为表达式,因为负号(-)在C语言中是一种单目运算符。但是,有些书上,也认为是常量。关于负的整数也是如此。本书认为负数为表达式。

[例 2.2] 浮点型常量的输出。

```
main()
{
    printf("%f\t%e\t%g\n",12e3,12.3,12.3);
    printf("%f\t%e\t%g\n",12.3456,12.3456,12.3456);
    printf("%d\t%.2f\t%f\n",12.3,12.345,123);
}
```

分析上述程序的输出结果,并上机验证。

2.1.2 字符常量和字符串常量

1. 字符常量

字符常量由单引号括起的一个字符组成的。例如,

'B', 'd', '9', '\n', '\0', '\\' 等

都是字符常量,单引号是字符常量的定界符。单引号符可由\"表示,即使用了转义序列的形式。因此,在字符常量中,对一般可打印字符采用直接写出字符符号的方法,而对不可打印字符采用转义序列的方法。

在C语言中,字符常量具有数值,该值便是该字符的ASCII码值。因此,一个字符常量可以像整数一样的参与一些运算,如加法、减法等运算。例如,

'd'-1

表示字符d的ASCII码值减去1,其差值为99。又例如,

'C'-'A'+ 'a'

表示将字符 C 的 ASCII 码值减去字符 A 的 ASCII 码值,再加上字符 a 的 ASCII 码值,其结果为 67,而此值正是大写字母 C 的 ASCII 值。

[例 2.3] 字符常量的输出。

```
main()
{
    printf("%c,%d\n",67,'n');
    printf("%c,%c,%c\n",'a','\b','b');
    printf("%c,%c,%c\n",'a','\r','b');
    printf("%c%c",'a','\007');
}
```

下面分析该程序的输出结果,进一步搞清一些转义序列表示的字符的使用方法。

该程序中有 4 个 printf() 函数的语句,依次执行输出如下:

第一个 printf() 函数的语句输出为:

C, 109

因为 67 所对应的是大写字母 C 的 ASCII 码值,因此,按 %c 格式输出 67 时,则为字母 C。又因为小写字母 m 的 ASCII 码值为 109,因此,按 %d 格式输出字母 m 的 ASCII 码值应是 109。

第二个 printf() 函数的语句输出为:

a, b

因为按 %c 输出字符常量 'a' 时,屏幕应显示 a,然后控制串中有一般字符逗号(,)则照样输出,这时显示为 a,。接着,再按 %c 输出 '\b',这是一个转义序列表示的字符,即退格符,在输出这个字符后,屏幕上显示为 a,接着又输出控制串中的第二个逗号,这时显示为 a,。再按 %c 输出字符常量 'b',因此,屏幕上应显示出 a,b。由于控制串中最后一个字符是 '\n',所以,光标被移到下一行第一个字符的位置。

第三个 printf() 函数的语句输出为:

,b

因为按 %c 输出字符常量 'a',又输出控制串中第一个逗号后,显示为 a,。再按 %c 输出字符常量 '\r' 后,光标被移到该行的首列,即字符 a 处,这时输出控制串中第二个逗号时,将字符 a 改为逗号(,)。接着,按 %c 输出字符常量 'b' 时,屏幕上显示为,b。再将光标移至下行首列。

第四个 printf() 函数的语句输出为:

两声鸣铃

因为按 %c 输出字符常量 '\a',则是机器的一声鸣铃,再按 %c 输出字符常量 '\007',则机器又一次鸣铃。这时,光标仍停留在该行的首列,即光标并没有移动。

2. 字符串常量

字符串常量是用一对双引号括起来的字符序列。双引号(" ")作为字符串常量的定界符,因此,在字符串中表示双引号应使用转义序列 '\"' 来表示。关于字符串与字符常量的区别在前面已经讲述过了,这里不再重复。

关于字符和字符串这两种常量的区别还可以作下述描述:

字符实际上是一个整型数,而字符串实质上是地址值。

这句话前部分通过学习字符常量可以理解,字符的整型数就是该字符的 ASCII 码值。而后部分在学完“指针”一章后便可理解。

字符常量是用来给 char 型变量赋值的,而字符串常量却是用来给 char 型数组赋值的,因此,两者是不相同的。

[例 2.4] 字符串常量的输出。

```
main()
{
    printf("%s,%s\n","ok!","\good");
    printf("x\ty\b\bz\ta\n");
    printf("abc\tdef\r\n\tpq\n");
}
```

分析该程序输出结果如下:

第一个 printf() 函数语句输出结果为:

ok!, "good"

因为按 %s 输出字符串常量 "ok!", 则显示出 ok!, 接着, 控制串中的逗号被输出。然后, 又按 %s 输出另一个字符串常量 "\good", 这里有转义序列表示的字符 '\', 即双引号, 这个字符串被输出显示为 "good"。然后, 光标移至下行首列。

第二个 printf() 函数语句输出结果为:

x 0000000 z 00 a

因为该语句是在屏幕上显示 printf() 函数中控制串中所指定的字符串常量。该字符串常量中有可打印字符, 又有不可打印字符 '\t', '\a' 和 '\b', 它们分别是水平制表符、空格符和退格符。水平制表符的作用是用来向右“跳格”, 每次跳到下一个“输出位置”, 一般系统中指定一个“输出区”占 8 列, 第一个输出区占 1 至 8 列, 下一个输出区将从第 9 列开始, 直到 16 列, 依此类推。空格符是将光标右移一个字符, 退格符是将光标移到所在字符的前一个字符处。弄清这三个常用的转义序列表示的字符的功能后, 便不难分析该字符串常量的输出结果。首先, 在该行首列显示字符 x, 接着, 光标右移至第 9 列(首列为第一列)输出显示字符 y, 然后输出两个 '\b' 字符, 光标向左退两列, 即在第 8 列处, 这时输出字符 z, 再输出空格符, 将 y 变为空格符, 又输出一个空格符后, 输出字符 a, 于是屏幕上显示上述结果。

第三个 printf() 函数语句输出结果为:

mn 000000 p q f

因为先输出 a b c 字符后, 再输出 '\t', 这时光标移至第 9 列。接着, 输出 def, 光标在第 12 列。当输出字符 '\r' 后, 光标将被移至该行的首列。即 a 字符下, 再输出 mn 时, 将 a 变为 m, 将 b 变为 n, 又输出 '\t', 则将 c 变为空格符, 光标移至第 9 列字符 d 下, 输出 p 时, 将 d 变 p, 输出 q 时, 将 e 变 q, f 没有被改变, 仍然存留, 光标被移至下行的首列, 因为最后有一个 '\n'。

2.1.3 符号常量

C 语言中, 常常用一个标识符来代表一个常量, 称为符号常量。符号常量在使用之前要先定义, 定义格式如下:

```
#define <符号常量名>(<常量>)
```

其中, <符号常量名>用标识符, 习惯上用大写字母, <常量>可以是数字常量, 也可以是字符

常量。这实际上是一个宏定义命令,通过这个宏定义将常量定义为一个符号常量。在C语言程序中用符号常量代替常量,在编译时首先将符号常量被所定义的常量替换后才进行编译,这个过程称为宏替换。

采用符号常量具有下述几个好处:

(1) 书写简单不易出错。使用符号常量可以将复杂的常量定义为简明的符号常量,使得书写简单,而且不易出错。例如,

```
#define PI 3.14159265
```

这里,符号常量PI被定义为3.14159265,在程序中书写PI,显然比书写3.14159265要简明。

(2) 修改程序方便。采用符号常量会给修改程序带来方便。例如,在一个程序中使用了某个符号常量共10次,根据需要对这一常量值进行修改,这时只需在宏定义命令中对定义的常量值进行一次修改。否则,要在程序中出现这一常量的10处都进行修改,这不仅带来一定麻烦,同时又易于出错。

(3) 增加可读性和移植性。由于符号常量通常具有明确的含义,因此,一见符号常量便可知道所表示的常量意义,例如,在前面的宏定义命令中,很明显PI表示圆周率,即 π 。所以可读性好。使用符号常量可将程序中影响环境系统的参数,如字长等,定义在一个可被包含的文件中,在不同的环境系统下,通过修改包含文件中符号常量的定义值来达到兼容的目的,于是可提高程序的移植性。

C语言中,符号常量习惯用大写字母表示,而一般变量用小写字母,以示区别。

[例2.5] 符号常量的使用。

```
#define PI 3.14159265
#define R 3
main()
{
    double circumference,area;
    circumference=2.0*PI*R;
    area=PI*R*R;
    printf("circumference=%lf,area=%lf\n",circumference,area);
}
```

执行该程序输出结果如下:

```
circumference=18.849556, area=28.274334
```

2.2 变 量

变量是指在程序中其值发生变化的量。变量具有三要素:名字、类型和值。

2.2.1 变量的名字

C语言中,要求对程序中所有的变量都须“先定义,后使用”。定义或说明一个变量时,要给出该变量的名字。变量名字的起法同前面讲过的标识符。变量名一般用小写字母,也可以用大写字母或大写小写字母混用。给变量起名时应尽量考虑到在名字中体现出该变量的含义或使

用目的,以便提高可读性。变量名不要同系统的关键字相同,变量名的长度按标识符的规定,有些微机系统只识别前8个字符,多余的不能被识别。本书中的一些例子,为了简明,变量名只用了一个字符,而在实际应用程序中的变量名还应能做到“见名知意”为好。例如,下列的变量名都反映了一定的含义,它们是合法的变量名:

count, size_of_number, sum, area 等。

在实际程序中,如果使用了没有被定义或说明的变量名时,在编译中会出现报错信息。这样可以保证程序中变量名使用正确,从而避免了用户将变量名写错。

2.2.2 变量的类型

在定义或说明变量时,除了指出该变量的名字外,还要指出该变量的类型。每一个变量被指定为一个确定的类型,编译时系统便可为该变量分配相应的内存单元,内存单元的分配与类型有关。例如,在一般16位微机中,int型变量被分配给2个字节的内存空间,而double型变量则应分配给8个字节的内存空间。另外,变量被指定类型后,编译时便可根据其类型来检查该变量所进行的运算是否合法,如果发现有不法之处,应及时报错。例如,

```
float a, b, c;
c=a%b;
```

这时编译系统将会指出float型变量不可做求余数(%)的运算。

C语言中变量类型十分丰富,它不仅有基本数据类型,还有构造数据类型。在每一类中又包括了若干种不同类型。

1. 基本数据类型

基本数据类型包括:整型、浮点型和字符型。而整型又分为普通整型(简称整型)、长整型和短整型三种。浮点型又分为单精度浮点型和双精度浮点型两种。整型和字符型又分为有符号和无符号两类。综上所述,C语言中基本数据类型的种类及其说明符号如表2-1所示。

表 2-1 基本数据类型

类 别	名 称	全称类型说明符	缩写类型说明符
整型	短整型	short int	short
	整 型	int	
	长整型	long int	long
	无符号短整型	unsigned short int	unsigned short
	无符号整型	unsigned int	unsigned
	无符号长整型	unsigned long int	unsigned long
字符型	字符型	char	
	无符号字符型	unsigned char	
浮点型	单精度浮点型	float	
	双精度浮点型	double float	double

2. 构造数据类型

构造数据类型是指由若干个相同的或不同的基本数据类型变量按不同规律组合构造而成的。例如,数组是一个简单的构造数据类型,该数据类型是由数目固定类型相同的若干个变量有序的集合。数组在C语言中使用较多,后面将会详细讲解。

除数组之外,C语言中构造数据类型还有结构、联合和枚举。它们将在本书中“结构”、“联合和枚举”两章中讲述。

有的书中将指针作为一种类型,本书直接把指针看作一种特殊的变量进行讲解。

变量除了有数据类型以外,在C语言中还具有存储类,变量的存储类也分为自动的、寄存器的、外部的和静态的,关于变量存储类的详细讲述可见本书“函数和存储类”一章。这里不去介绍存储类,并规定在定义变量时,凡是在函数体内的不加存储类说明的都为自动类的。下面只讨论数据类型。定义或说明变量的数据类型的格式如下:

〈数据类型〉〈变量名表〉

〈变量名表〉是由相同类型的若干个变量名组成,多个变量名之间用逗号分隔。〈数据类型〉是类型说明符,用来指出所定义的变量的类型。

下面使用基本数据类型定义一些变量。例如,

```
int a, b, c;
```

定义或说明 a, b 和 c 三个变量是整型的。

```
float x, y;
```

定义或说明 x 和 y 两个变量是单精度浮点型变量。

```
double m, n;
```

定义或说明 m 和 n 两个变量是双精度浮点型变量。

```
char d, e, f;
```

定义或说明 d, e 和 f 三个变量是字符型变量。

```
long p, q;
```

定义或说明 p 和 q 是两个长整型变量。

```
unsigned u, v;
```

定义或说明 u 和 v 是两个无符号整型变量。

等等。

上述说明语句中,使用类型说明符来定义或说明一个或多个相同类型的变量,多个变量用逗号分隔。

关于构造数据类型变量的定义或说明将在本书中后面章节中介绍。

2.2.3 变量的值

一个变量在使用之前必须先定义或说明,系统给它们分配一定的存储单元,接着要给变量赋值或者赋初值。具有确定值的变量才能在表达式中使用。有些变量虽然被定义了,如果没有被赋值或被赋初值,则是无意义的,即不可使用。

在C语言中,赋值和赋初值尽管都可以使变量获取数据,但它们不是一回事。赋值是将一个数值送给一个变量,改变这个变量已有的值为所送给的值,于是变量将具有该值,直到下次被再赋值为止。赋值是使用赋值表达式进行的。赋初值是在定义或说明变量时,将一个数值送给变量的,使变量被定义后便有了该值,直到被改变为止。赋初值是在编译时执行的,而赋值是在运行时实现的,两者所占用的机器的时间不同。例如,

```
int a=5, b=8;
```

这是在定义 a 和 b 为 int 型变量的同时,便给 a 赋初值为 5,给 b 赋初值为 8。这是属于赋

初值。又例如，

```
int a, b;  
a=5;  
b=8;
```

这是先定义了 a 和 b 两个变量,都是 int 型的,定义时并没有给赋初值。接着,使用了两个赋值表达式语句给变量 a 赋值为 5,给变量 b 赋值为 8。

不同类型的变量要赋给与变量类型相对应的值。例如,

```
char c;  
float d;  
c='a';  
d=5.26;
```

因为变量 c 是 char 型的,因此,赋给一个字符 'a';而变量 d 是浮点型的,因此,赋给一个实数,这是正确的。如果所赋的值与变量类型不一致时,系统将进行转换,其原则是将所赋的值转换成被赋变量的类型。

给变量所赋的值是有一定范围的,这一范围的大小是由机器和变量类型决定的,在同一台机器上不同类型变量的取值范围是不同的。下面给出表 2-2,它表示出在一台 IBM PC 机(16 位)上不同数据类型所占的字节数(一个字节 8 位)和取值范围。

表 2-2 不同数据类型所占字节数和取值范围

类 型	所占字节数	取值范围
int	2	-32768~32767 ($-2^{15} \sim (2^{15}-1)$)
short	2	-32768~32767 ($-2^{15} \sim (2^{15}-1)$)
long	4	-2147483648~2147483647 ($-2^{31} \sim (2^{31}-1)$)
unsigned	2	0~65535 ($0 \sim (2^{16}-1)$)
unsigned short	2	0~65535 ($0 \sim (2^{16}-1)$)
unsigned long	4	0~4294967295 ($0 \sim (2^{32}-1)$)

在实际程序中,一个变量值超出它所对应的数据范围,就将产生溢出,溢出的数据是没有的。在编写程序时,要防止其结果产生溢出。

在 IBM PC 的 16 位微机中,单精度浮点数占 4 个字节(32 位),可提供 7 位有效数字,取值范围可在 $10^{-38} \sim 10^{38}$ 之间。双精度浮点数占 8 个字节(64 位),可提供 15~16 位有效数字,其取值范围约为 $10^{-308} \sim 10^{308}$ 。可见, float 类型和 double 类型的变量都是用来存放浮点数的,其区别在于 double 型所存放数值的有效位数比 float 型的要高一些。例如,

```
double x;  
x=12345.6789;
```

由于 x 是 double 型的,它可接收 9 位数字并存放在 x 中。如果将 x 定义为 float 型的,这时 x 只能接收 7 位有效数字,则最后两位小数将不起作用。

[例 2.6] 变量的定义和赋值。

```
main()  
{
```

```

int a;
char c='B',bell='\007';
float x=3.25;
a=65;
printf("%c,%c\n",c,c-1);
printf("%d,%c,%c\n",c,a,bell);
printf("%.2f,%.2f\n",x,x+x);
}

```

该程序执行后,输出结果如下:

```

B, A
66, A, <鸣铃>
3.25, 6.500000

```

2.3 数 组

数组是 C 语言中常用的一种数据类型,它是一种简单的构造数据类型。这里只介绍关于数组的一些基本概念,关于数组在 C 语言程序中的使用将在本书“指针”一章中讲解。

2.3.1 数组的定义

数组是数目固定类型相同的若干变量的有序集合。数组这种类型的特点是组成数组的若干变量的类型必须相同,例如,int 型数组中的若干元素必须都是 int 型。另外,数组的若干个元素在内存中是按一定顺序存放的,因此,它是有序的。

数组的定义格式如下:

〈类型说明符〉〈数组名〉[〈常量表达式 1〉][〈常量表达式 2〉]…

其中,〈类型说明符〉是用来说明数组中各个元素的类型,应该包括数据类型和存储类,不加存储类说明符的为外部的或自动的。〈数组名〉用来标识数组的标识符,数组名定义规则同标识符。〈常量表达式 1〉、〈常量表达式 2〉…用来表示某维数组的元素个数。数组的维数由〈数组名〉后面方括号和括起的常量表达式的个数决定。有一个方括号和括起的常量表达式称为一维数组,有二个方括号和括起的常量表达式称为二维数组,依次类推,有 n 个方括号和括起的常量表达式的称为 n 维数组。常量表达式可包含常量和符号常量,不包含变量。例如,

```
int a[8];
```

a 是一个一维数组的数组名,该数组是由 8 个 int 型元素组成的。又例如,

```
char b[5][3];
```

b 是一个二维数组的数组名,该数组是由 15 个(5×3)char 型元素组成的。又例如,

```
float c[2][3][4];
```

c 是一个三维数组的数组名,该数组是由 24 个(2×3×4)float 型元素组成的。

数组元素的表示格式如下:

〈数组名〉[〈下标 1〉][〈下标 2〉]…

其中,〈下标 1〉、〈下标 2〉、…可以是整型常量表达式或整型表达式。C 语言中规定下标从 0 开始。例如,int m[10]中表示数组 m 有 10 个元素,每个元素是 int 型,这 10 个元素分别表示为

`m[0],m[1],m[2],m[3],m[4],m[5],m[6],m[7],m[8],m[9]`。而 `m[10]` 将不是该数组的元素。又例如

```
char n[3][2];
```

表示 `n` 是一个二维数组,共有 6 个元素,每个元素是 `char` 型的,这 6 个元素是: `n[0][0],n[0][1],n[1][0],n[1][1],n[2][0],n[2][1]`。`n` 数组可以看作是一个 3 行 2 列的数组(3×2)。进一步还可以把“行”看作是一种特殊的一维数组,它的每个元素(即“列”)又是一个一维数组,因此,二维数组实质上是一维数组的一维数组。例如,数组 `n` 有 3 行 2 列,可将 `n` 看成为一维的“行”数组,它有 3 个元素: `n[0],n[1]` 和 `n[2]`,这是“行”数组的 3 个元素。每个元素又是一个包含有 2 个元素的一维数组。这里,可以认为 `n[0],n[1]` 和 `n[2]` 是 3 个一维数组的名字。

又例如,

```
int p[2][3][4];
```

表示 `p` 是一个三维数组,共有 24 个元素,每个元素是 `int` 型的,这 24 个元素是: `p[0][0][0],p[0][0][1],p[0][0][2],p[0][0][3],p[0][1][0],p[0][1][1],p[0][1][2],p[0][1][3],p[0][2][0],p[0][2][1],p[0][2][2],p[0][2][3],p[1][0][0],p[1][0][1],p[1][0][2],p[1][0][3],p[1][1][0],p[1][1][1],p[1][1][2],p[1][1][3],p[1][2][0],p[1][2][1],p[1][2][2],p[1][2][3]`。

一般常用的数组有一、二、三维,更高维数组很少使用。

数组的上述引用方法被称为数组元素的下标表示法。以后还将会看到数组元素的指针表示和指针、下标混合表示法。

数组的各个元素存放在内存单元中是按一定顺序的。上面讲到的各维数组的若干个元素的表示时,就是按照其在内存中存放的顺序给出的。对于一维数组,则按其元素的下标由小到大的顺序存放;对于二维数组,则按行的顺序存放,即先存放第一行的元素,再存放第二行的元素,依此类推,由于每一行是一个一维数组,则每行的各个元素按一维数组存放规则存放;对于三维数组和三维以上的数组,则按第一维的下标变化最慢,最右边维的下标变化最快的原则进行排序。以三维数组为例,正如前边给出的 `p` 数组的 24 个元素的顺序。

2.3.2 数组的赋值

数组的赋值分为赋初值和赋值两种。赋初值又称为初始化,是在编译时进行的,故不占用运行时间。赋值是用赋值表达式语句,在运行时间进行的。二者虽然都可使数组元素获得所需要的值,但是就其方法和时间是不同的。

1. 数组的赋初值

数组被赋初值不是所有数组都可做到的。C 语言规定:只有静态存储类(`static`)和外部存储类(`extern`)的数组才能被赋初值。

赋初值的方法是使用一种称为初始值表的方法。该方法是在定义或说明数组时用一对花括号将要赋给数组各元素的值括起来,按其顺序赋给该数组。

对一维数组赋初值方法如下:

```
static int a[5]={5,4,3,2,1};
```

经过这样的初始化后,使得数组 `a` 的 5 个元素(按顺序)分别获得的值是:5,4,3,2,1。即 `a[0]` 获得 5,`a[4]` 获得 1。使用这种办法也可以使一个数组中的某些元素获值,例如,

```
static int b[8] = {1,3,5,7,9};
```

这里,数组 b 的 8 个元素中,前面的 5 个元素 b[0],b[1],b[2],b[3],b[4]分别获得的值为 1,3,5,7 和 9。而 b 数组中 b[5],b[6],b[7]这三个元素没有被赋初值。

在使用初始值表的方法赋初值时,需要注意的是要使初始值表中数据项的个数小于或等于而不得大于待初始化的数组元素的个数,这就保证了在给数组赋初值时不会使数组产生越界。为了使得初始表中的所有数据项正好赋给待初始化的数组元素,可在初始化时不给定数组的大小。例如:

```
int m[] = {1,2,3,4,5};
```

等价于

```
int m[5] = {1,2,3,4,5};
```

在省略数组大小时,系统将会根据初始值表中数据项的多少来自动确定数组的大小,在上例中,由于数据项为 5,因此,数组的大小也为 5。如果用初始值表给一个数组中的部分元素赋值时,待初始化的数组的大小就不能省略。例如,

```
static int a[8] = {1,2,3,4};
```

这里,数组 a 的大小 8 就不能省略。上述语句将给数组 a 的前 4 个元素赋了初值,而后 4 个元素没有赋初值,即保持其缺省值,皆为 0。

对二维数组赋初值方法如下:

二维数组赋初值的原则与一维数组相同,同样要求初始值表中数据项的个数要小于或等于待初始化的数组元素的个数。使用初始值表进行赋初值就是使得待初始化的数组中的全部或部分元素按其顺序从初始值表中对应项获取值。例如,

```
int a[2][3] = {1,2,3,4,5,6};
```

表示数组 a 的 6 个元素按其内存存放顺序依次获得 1,2,3,4,5,6 各值。对二维数组也可以按行初始化。例如,

```
int a[2][3] = {{1,2,3},{4,5,6}};
```

在初始值表中,对每一行所对应的数据项用一个花括号括起来,这样看起来比较直观,该例中,将{1,2,3}赋给第一行的 3 个元素,再将{4,5,6}赋给第二行的 3 个元素。

也可以对二维数组的部分元素进行初始化。例如,

```
int b[3][4] = {1,2,3};
```

表示对数组 b 的前 3 个元素 b[0][0],b[0][1]和 b[0][2]赋初值分别为 1,2 和 3。又例如,

```
int b[3][4] = {{1},{2},{3}};
```

表示对数组 b 的每一行的首元素赋初值,即对 b[0][0]赋初值为 1,对 b[1][0]赋初值为 2,对 b[2][0]赋初值为 3。而数组 b 的其余 9 个元素都为缺省值 0。又例如,

```
int b[3][4] = {{},{0,1},{1,2,3}};
```

表示对数组 b 的第一行不赋初值,对第二行的 b[1][0]和 b[1][1]分别赋初值为 0 和 1,对第三行的 b[2][0],b[2][1]和 b[2][2]分别赋初值为 1,2 和 3,数组 b 的其余元素值为 0。

如果对二维数组的全部元素用初始值表进行赋初值时,则所定义的数组的第一维的大小可以不指定,但是第二维的大小必须指定。例如,

```
int b[][3]={1,2,3,4,5,6};
```

系统将会计算出数组 b 的第一维的大小是 2,并自动确定。

对三维数组的初始化的方法同于前面所述,不再重复,仅举一例,例如,

```
int c[2][3][4]={{ {1,2,3,4},{5,6,7,8},{9,10,11,12}},{ {13,14,15,16},
                  {17,18,19,20},{21,22,23,24}}};
```

等价于 `int c[2][3][4]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};`

关于部分元素的赋值方法和省略数组第一维大小的方法与二维数组相同。

2. 数组的赋值

数组的赋值,实际上是对数组的各个元素的赋值。对数组的各个元素赋值可通过赋值表达式语句进行。例如,

```
int a[5],b[2][3];
a[0]=a[1]=a[2]=1;
a[3]=a[4]=2;
b[0][0]=3;
b[0][1]=4;
...
```

对数组的赋值也可采用循环的方法,关于循环语句将在后面会讲解。例如,

```
for(i=0; i<5; i++)
    a[i]=i+1;
```

表示对数组 a 的 5 个元素分别赋值为 1,2,3,4 和 5。

同样,对二维数组赋值可采用二重循环。值得说明的是采用循环的办法赋值要求数组元素值之间存在某种可使用循环的关系。关于二、三维数组用循环赋值的例子将会在后面的程序中看到,这里不再举例。

这里需要说明的是不能用初始值表的方法给数组赋值。下面作法是错误的。

```
int a[5];
a={1,2,3,4,5};
```

给数组赋值只能是对数组的各个元素赋值。

[例 2.7] 一维数组的赋值和引用。

```
main()
{
    static int a[5]={9,7,5,3};
    a[2]=10;
    printf("%d,%d,%d\n",a[0],a[2],a[4]);
}
```

执行该程序输出结果如下:

```
9,10,0
```

因为通过对数组 a 的初始化,使 a[0]为 9,a[2]为 5,a[4]为缺省值 0。又经过对 a[2]的赋值,将 a[2]的值改变为 10。所以,输出如上的结果。

[例 2.8] 二维数组的赋值和引用。

```
main()
{
    int i,s=0;
    static int b[3][3]={ {6},{7,8}};
    for(i=0;i<3;i++)
        b[2][i]=i+1;
    for(i=0;i<3;i++)
        s+=b[i][i];
    printf("%d\n",s);
}
```

执行该程序输出结果如下：

17

因为该程序中,通过对数组 b 赋初值使得 b[0][0]获值为6,b[1][0]获值为7,b[1][1]获值为8,其余元素为0.再通过程序中的循环赋值使得 b[2][0]获值为1,b[2][1]获值为2,b[2][2]获值为3.程序再使用一次 for 循环,求得二维数组的主对角线各元素之和,这时 b[0][0]为6,b[1][1]为8,b[2][2]为3,所以求和为17.于是,输出该和 s 的值为17。

2.3.3 字符数组

前面讲过的数组都是数字数组,即数组元素都是数字型的。除了数字数组之外,在 C 语言中,常用的还有字符数组。所谓字符数组是指数组的各个元素是 char 型的数组。

字符数组的定义和赋值与数字数组相同,只是为了字符数组赋初值的方便,C 语言规定可将一个字符串常量作为初始值表赋初值给字符数组的各个元素。关于赋初值和赋值的具体规则与前面讲的相同。例如,

```
static char m[5]={ 'a','b','c','d','\0' };
```

于是,字符数组 m 存放着 abcd4个字符和一个字符串结束符 '\0'。它等价于

```
static char m[5]="abcd";
```

这里,数组 m 有5个元素,字符串常量中只有4个字符,而结束符 '\0' 是系统自动加上的。它还可以写成如下格式:

```
static char m[]="abcd";
```

如果写成下面格式:

```
static char n[6]="xyz";
```

表示字符数组 n 中有6个元素,前3个元素被赋初值 'x','y' 和 'z',第3元素(以第0个元数起)为 '\0',其余元素为空。

在用字符串常量给字符数组赋初值时,同样要求字符串常量中字符的个数加1要小于或等于字符数组的元素个数。为了不写成“越界”赋初值的形式,可将字符数组(一维)的大小省略。例如,

```
static char n[]="xyz";
```

这表示 n 是一个字符数组,它有4个元素,分别是 'x','y','z' 和 '\0'。

关于给二维的字符数组赋初值可用初始值表的方法,给出字符数据项,也可以用字符串常

量。因为二维数组可以看成是若干个一维数组构成的,每个一维的字符数组可以使用字符串赋初值。例如,

```
static p[3][4]={{'a','b','c','\0'},{'m','n','p','\0'},{'x','y','z','\0'}};
```

它等价于:

```
static p[3][4]={"abc","mnp","xyz"};
```

显然后者比前者更加简明。或者写成:

```
static p[][4]={"abc","mnp","xyz"};
```

这里,数组 p 的第一维的大小省略了,这是可以的,系统会自动确定。

可见,二维的字符数组可以用来存放若干个字符串。所存放字符串的多少,则为二维字符数组第一维的大小,所有存放的字符串中最长的个数加1,则为二维字符数组第二维的大小。

字符数组同样也有三维的,它可以理解为用来存放若干组字符串,每组又有若干个。

这里值得注意的一点是字符数组与字符串的不同。一维的字符数组可以用来存放一个字符串,也可以用来存放一些字符。一个字符数组中存放的是否是字符串,关键取决于给它赋值或赋初值的情况。如果对一个字符数组在赋值或赋初值时将字符 '\0' 赋给了某个元素,则该字符数组中存放的是一个以 '\0' 结束的字符串。例如,

```
static char x[3]={'a','b','\0'};
```

```
static char y[3]={'a','b','c'};
```

这里,数组 x 中存放的是一个字符串"ab",而数组 y 中存放的不是字符串,而是3个字符 a,b,c,因为数组 y 的3个元素中没有结束符 '\0'。

如果将一个字符串常量在定义数组时赋初值给该数组,则该数组将存放所赋予的字符串,这时,系统将自动给数组加一个 '\0' 作为结束符。应注意字符串的有效字符个数加1后应小于或等于字符数组的元素个数,或者在赋初值时不确定字符数组的大小,例如,

```
static char x[5]="good";    或者
```

```
static char x[]="good";
```

C 语言中规定了字符串的结束符后,在程序中判断字符串是否结束时不是采用该字符串的长度而是根据检测字符串中的结束符 '\0',这将给字符串的判断带来一些方便。需要说明的是,结束符 '\0' 是指 ASCII 码值为0的那个“空字符”,用它作结束符只是起到一个辨认标志的作用,而不会增加有效字符的个数。

有关字符串处理函数将在本书“指针”一章中讲解,同时还要讲解由于字符指针的引入给字符串处理带来了方便。

[例2.9] 字符数组的赋值和输出。

```
main()
{
    int i;
    static char a[]="good mornning!";
        b[]={'a','b','c','d','e','\0'};
    for(i=0;i<14;i++)
        printf("%c",a[i]);
    printf("\n");
    printf("%s\n",b);
}
```

执行该程序输出结果如下：

```
good ☐ mornning!  
abcde
```

通过该例将字符串的输入输出方法总结如下：

(1) 字符串的输入方法可以单个字符输入，例如，本例中 b 数组；也可以直接输入一个字符串，例如，本例中 a 数组。

(2) 字符串的输出方法可以单个字符输出，使用格式符 %c，最终形成字符串形式，例如，本例中 a 数组的输出；也可以使用格式符 %s，一次输出字符串，例如，本例中 b 数组。

(3) 使用格式符 %c 输出一个字符时，对应参数是数组元素名；使用格式符 %s 输出一个字符串时，对应参数是数组名，实际上是一个地址值。

(4) 在使用格式符 %s 输出一个字符串时，实际上是从给定的地址开始逐一输出其存放的字符直到遇到 '\0'，则结束输出。在数组长度大于字符串的实际长度时，也是只输出到 '\0' 为止。在一个存放有多个字符串的一维数组中，也是在遇到第一个 '\0' 时结束输出。

[例2.10] 二维字符数组的输入和输出。

```
main()  
{  
    int i,j;  
    static char x[3][6]={"black","red","blue"},y[3][6];  
    printf("Input strings: ");  
    scanf("%s%s%s",y[0],y[1],y[2]);  
    printf("Print strings:\n");  
    for(i=0;i<3;i++)  
        printf("%s\n",x[i]);  
    printf("%s,%s,%s\n",y[0],y[1],y[2]);  
}
```

执行该程序后，屏幕显示如下信息：

Input strings: Wang ☐ Zhang ☐ Zheng ✓

键入由空格分隔的3个字符串，并回车后显示如下信息：

```
Print strings:  
black  
red  
blue  
Wang, Zhang, Zheng
```

通过该例使读者对二维字符数组的输入和输出方法有所了解，请读者总结一下你有哪些收获。

提示：对字符数组可以通过赋初值的方法进行输入，还可以使用 scanf() 函数输入字符串。

2.4 类型转换

C 语言中,类型的转换不是很严格的,它包含了自动转换和强制转换两种方式。

2.4.1 自动转换

自动类型转换是指系统将根据指定的规则自动地进行数据类型的转换。

(1) 隐含的自动转换。这种转换包含以下两点:

① char 型和 short 型转换为 int 型。

② unsigned char 型和 unsigned short 型转换为 unsigned 型。

(2) 所有的浮点数运算都以双精度进行。即 float 型转换为 double 型。

(3) 各类数值型数据间的混合运算时,系统自动将参与运算的各类数据转换为它们之间数据类型中最高的类型。所谓数据类型的高低是指该数据类型在内存中所占字节数的多少,占字节数多的就意味着数据类型高,否则认为数据类型低。数据在内存中所占字节的字节数又称为数据长度,占的字节数越多,就说该数据的长度越长。因此,数据类型的高低将由该数据长度所定,数据长度越长,则其类型越高。

C 语言中,各类数值型数据类型的高低顺序排列如下:

int \rightarrow unsigned \rightarrow long \rightarrow double

该顺序自左至右数据长度增加,类型增高。该顺序中,没有包含 char, short 和 float 型,而实际上,前面已讲过 char 型应先转换为 int 型,short 型也转换为 int 型,float 型转换为 double 型。例如,

```
float x;  
double y;  
5 + 'm' - x/2 + y * 5;
```

在计算该式子的值时要进行类型转换。按运算顺序:① 在作 $x/2$ 的运算时,将整数 2 转换为 float 型,其结果为 float 型。② 在作 $y * 5$ 的运算时,将整数 5 转换成 double 型,其结果为 double 型。③ 在进行 $5 + 'm'$ 运算时,'m' 转换成 int 型,其结果为 int 型。④ 将③的结果与①的结果相减时,先将③的结果转换为 float 型,其结果为 float 型。⑤ 将④的结果与②相加时,先将④的结果转换成 double 型,其结果为 double。该表达式计算的最终结果为 double 型。

(4) 在赋值表达式中,赋值号右边的表达式的类型被系统自动转换为赋值号左边变量的类型。例如,

```
short a;  
int b;  
long c;  
c = a + b;
```

在执行 $c = a + b$ 时,先将变量 a 转换为 int 型后,再与变量 b 相加,其和为 int 型。再将其 int 型和转换为 long 型,并赋值给变量 c,因为 c 是 long 型的。

在这种赋值转换中,当右值(指赋值号右近的表情式)精度高而左值(指赋值号左边的变量)精度低时,在由高精度转换为低精度的过程中会引起数据精度的下降,有时将影响其运算

结果。

(5) 在带有表达式的返回语句中,C 语言规定要将表达式的类型转换为函数的类型(即返回值的类型)后再将其值返回给调用函数。详见本书“函数和存储类”一章。

2.4.2 强制转换

类型的强制转换是在一个表达式前加一个强制转换类型的运算符,将表达式的类型强制转换为所指定的类型。具体格式如下:

`(<类型说明符>)(表达式)`

表示将<表达式>的类型强制转换为圆括号中所指定的<类型说明符>的类型。这里,(<类型说明符>)是一种运算符,将在下一章中讲解。例如,

```
(double)(a+5);
```

假定,a 是 int 型变量,表达式 a+5 的值仍然是 int 的,通过强制转换符后,将表达式 a+5 的类型转换成为 double 型。

这种强制转换并不改变原来变量(如 a)的类型,只是暂时的一次性转换。

强制类型转换是很有用的,以后程序中会常见到。例如,使用 sqrt() 函数来求一个数的平方根,该函数要求被开方的数是 double 型的。如果求 int 型数 49 的平方根,可使用如下式子:

```
int a=49, s;  
s=(int) sqrt((double) a);
```

这表明为求 int 型数 49 的平方根,先将 a 转换为 double 型,这是 sqrt() 函数的要求,然后再将开平方后的结果强制为 int 型赋给 s。这里,不用(int)也可以,因为 s 是 int 型变量,赋值时会自动转换为 int 型。

练 习 题

1. 什么是常量? C 语言中有哪些种常量?如何从数值上表示出不同种类的常量?
2. 整型常量有哪些种?浮点型常量如何表示?
3. 字符常量和字符串常量有哪些不同?
4. 什么是符号常量?使用符号常量有哪些好处?如何定义符号常量?
5. 什么是变量?变量有哪三要素?C 语言中对变量名有何要求?
6. C 语言中变量类型有哪些?常用的基本数据类型有哪些?它们的类型说明符是什么?
7. 变量的赋初值和赋值有什么不同?
8. 什么是数组?如何定义一个数组?
9. 什么是数组元素?数组元素如何表示?数组元素在内存中是按怎样的顺序存放的?
10. 什么是数组的大小?什么是数组的下标?二者有何区别?
11. C 语言中规定数组下标是从 0 开始还是从 1 开始?数组下标可以是一个表达式吗?
12. 任何数组都可以赋初值吗?任何数组都可以赋值吗?
13. 在使用初始值表给数组赋初值时应注意什么问题?
14. 如何给一个数组的部分元素赋初值?试举一、二维数组的例子进行说明。
15. 什么是字符数组?如何将一个字符串常量赋给字符数组?
16. 字符数组中一定是存放字符串吗?并举例说明之。

17. 字符串一定要在存放时加结束符 '\0' 吗?这样做会带来什么方便?
18. C 语言中类型自动转换的规则有哪些?
19. 数据类型的高低指的是什么?类型转换是否都是从低类型转换为高类型?
20. 如何实现类型的强制转换?

作 业 题

1. 判断下列描述是否正确,对者划 √,错者划 ×。

- (1) 变量名可使用字母、数字和部分特殊字符如 '*'、'?' 和 ':' 等。
- (2) C 语言中,字符串常量可与整数作加减操作。
- (3) C 语言中,整型数可分为长整型、短整型和普通整型,它们之间的区别仅在于数据长度不同。
- (4) 变量的赋值和赋初值是一回事。
- (5) C 语言中规定数组元素的下标是从 0 开始的。
- (6) 数组的各个元素不一定是连续地按顺序存放在内存中。
- (7) 在使用的初始值表给数组元素赋初值时,只允许对所有元素赋初值,不能对部分元素赋初值。
- (8) 一维字符数组用来存放字符串时,数组的大小一定要比字符串中实际字符个数大 1。
- (9) 强制类型转换中只能允许低类型转换为高类型,否则将影响精度。
- (10) 一个表达式的类型取决于组成它的各操作数中类型高的一个的类型。

2. 选择填空

- (1) 下列常量中,() 是字符常量。
(a) 5 L (b) 0 1 2 (c) 'm' (d) "a"
- (2) 下列字符集中,() 可作为变量名。
(a) a-b (b) int (c) x*y (d) aA
- (3) 指出下列浮点数的表示形式中,() 是错误的。
(a) e5 (b) 12.0E-1 (c) 5.67 (d) 0.0
- (4) 下列的字符常量,() 是非法的。
(a) '\a' (b) 'b' (c) '\ ' (d) '3'
- (5) 下列各种给字符数组赋初值的语句中,() 是错误的。
(a) static int a[5]={1}; (b) static char b[3]="abc";
(c) static char c[]="abc"; (d) static int a[]={1,2,3,4,5,6};

3. 按照下列描述写出相应的语句来。

- (1) 给两个 int 型变量 s 和 b 赋值分别为 5 和 6。
- (2) 给一个一维字符数组 m 赋值一个字符串 "abc"。
- (3) 给一个一维字符数组 x 赋初值,该数组大小为 10,所赋的初值为 'a', 'b' 和 'm'。
- (4) 使用 scanf() 函数给一个二维字符数组赋字符串。已知二维数组的第一维大小为 2,第二维大小为 15,所赋的字符为 "computer" 和 "programming"。
- (5) 定义符号常量 N 的值为 100。

4. 分析下列各程序的输出结果。

(1)

```
main()
{
    int a,b;
    float m,n;
```

```

a--b=5;
m=1.5;
n=7.5;
printf("%d\n",a+b);
printf("%.2f\n",n--m);
}

```

(2)

```

main()
{
    char a,b;
    a='\n';
    b='a';
    printf("%c,%c,%c\n",65,a,b);
}

```

(3)

```

main()
{
    static char m[]="This is a program.";
    printf("%s\n",m);
    printf("abc\babc\tabc\rabc\n");
}

```

(4)

```

main()
{
    static int a[3][3]={0},{1,2,3},{5};
    printf("%d,%d,%d\n",a[1][1],a[3][0],a[0][0]+a[1][0]+a[2][0]);
}

```

(5)

```

main()
{
    int i;
    static char x[]="abcdefg";
    printf("%c",x[6]);
    for(i=5;i>=0;i--)
        printf(" * %c",x[i]);
    printf("\n");
}

```

第三章 运算符和表达式

本章讲述 C 语言中另一类基础内容:运算符和表达式。C 语言中运算符十分丰富,包括40多种不同的运算符,分为15种优先级和两类结合性。C 语言被称为多表达式语言,这是因为 C 语言中表达式的种类多,C 语言程序中表达式语句多。本章着重讲解各种运算符的功能、使用方法、优先级和结合性;进一步讲解各种表达式的求值方法和类型,指出在表达式求值中应该注意的若干问题。本章内容将为下一章“语句”打下基础。

3.1 常用运算符的功能

本节将详细讲述常用的运算符的功能。由于 C 语言中运算符种类多、功能强,按其用处分类逐一讲述如下。

3.1.1 算术运算符

算术运算符分单目运算符和双目运算符两种。单目运算符只有求负数运算符(一),又称为负值运算符。与它对应的正值运算符(+)很少使用,这里不详述。负值运算符是用来改变一个操作数的正或负号的。在一个正数前加一负值运算符后,则该数变为负数;在一个负数前加一负值运算符后,则该数变为正数。

双目运算符有如下5种:

+:加法运算符。如,3+9+11。

-:减法运算符。如,7-4。

*:乘法运算符。如,5*3。

/:除法运算符。如,18/6。

%:求余运算符。如,12%5。

这5种运算符都要求有两个操作数,故称双目运算符。

说明:

(1) 除求余运算符只适用整型数运算外,其余运算符可以作整数运算,也可以作浮点数运算。加、减法运算符还可作字符运算。

(2) 两个整数相除其结果为整数。例如,8/5结果为1,小数部分舍去。如果两个操作数有一个为负数时,则舍入方法与机器有关。多数机器是取整后向零靠拢。例如,8/5取值为1,-8/5取值为-1,但也有的机器例外。

(3) 求余运算符的功能是舍掉两整数相除的商,只取其余数。两个整数能够整除,其余数为0。例如,8%4的值为0。当两个整数中有一个为负数,其余数如何处理呢?请记住,按照下述规则处理:

$$\text{余数} = \text{被除数} - \text{除数} * \text{商}$$

这里,被除数是指%左边的操作数,除数是指%右边的操作数,商是两整数相除的整数商。

例如, $-8/5$ 的余数应该是

$$-8-5 * (-1) = -3$$

而 $8/-5$ 的余数应该是

$$8-(-5) * (-1) = 3$$

(4) 一个字符常量可与整数作加减运算。下列表达式是合法的:

`c+'A'-'a'`

其中, `c` 是一个字符变量, 该表达式将 `c` 所存放的大写字母变成了小写字母。

3.1.2 增1和减1运算符

增1和减1运算符也是属于算术运算符, 这是两个单目运算符, 这两个运算符具有副作用, 因此, 单独列出讲解。

增1运算符的作用是使被作用的变量值增1, 而使其表达式的值或增1或不增1, 这取决于前缀作用还是后缀作用。前缀作用时, 则表达式值增1; 后缀作用时, 则表达式值不增1。所谓前缀作用是指运算符作用在变量之前, 后缀作用是指运算符作用在变量之后。增1运算符是由两个加号组成的: `++`。例如:

```
int i=5;
```

执行 `++i` 后, `i` 的值为6, 而 `++i` 的值也是6。执行 `i++` 后, `i` 的值为6, 而 `i++` 的值为5。

可见, 增1运算符实际上有两个作用, 除了可以产生一个表达式的值之外, 还会改变其变量本身的值。一般的运算符只有产生表达式的值这一功能, 而没有改变变量值的功能。例如, 求负数运算符, 在下述表达式中, 没有改变变量本身的值:

```
int i=5;
```

`-i` 的值为-5, 而 `i` 的值仍然是5。

我们说增1运算符有“副作用”是指这种运算符除了产生表达式值的作用之外, 还有一个改变变量的作用, 后一个作用不是所有运算符都有的, 故称为“副作用”。在 C 语言中, 具有副作用的运算符除了增1减1运算符外, 还有赋值运算符。

总结一下增1运算符功能上的特点如下:

增1运算符作用于一变量, 可使该变量的值增1; 如果前缀作用于变量, 则其表达式的值为原变量值增1; 如果后缀作用于变量, 则其表达式的值为原变量的值。

同样的道理, 减1运算符功能上的特点如下:

减1运算符作用于一变量, 可使该变量的值减1; 如果前缀作用于变量, 则其表达式的值为原变量值减1; 如果后缀作用于变量, 则其表达式的值为原变量的值。例如:

```
int a=2;
```

执行 `--a` 后, `a` 变量的值为1, `--a` 表达式的值为1; 执行 `a--` 后, `a` 变量的值为1, `a--` 表达式的值为2。

在学习和掌握增1和减1运算符时, 应该搞清楚下列的两个不同。

- 变量值和表达式值的不同;
- 前缀作用和后缀作用的不同。

在使用增1和减1运算符时应该注意: 它只能作用于变量, 而不能作用于常量和表达式, 例如, 下列写法都是不合法的:

```
int a=5, b=3;
```

++-a, (a+b)++, ++10, --(a*b)等

关于增1减1运算符组成表达式计算问题在本章后面还会讲述。

3.1.3 关系运算符

关系运算符都是双目运算符,其功能是用来对两个操作数的大小进行比较的。C语言提供了如下6种关系运算符:

<: 小于运算符。如, $a < b$ 。

<=: 小于等于运算符。如, $c \leq 5$ 。

>: 大于运算符。如, $b > c$ 。

>=: 大于等于运算符,如, $b \geq 0$ 。

==: 等于运算符。如, $c == b$ 。

!=: 不等于运算符。如 $c != 10$ 。

在这6个运算符中,前4个优先级相同,后两个优先级相同。

关系运算符组成的关系表达式的值是逻辑值,即是真或假。例如, $a > 5$ 的值要么是真,要么是假,取决于 a 的值。如果 a 值为8,则 $a > 5$ 为真。在C语言中没有逻辑类型的量,规定“真”用1表示,“假”用0表示。于是, $a > 5$ 的值为1。这里的1就是数字1。例如,下述表达式是合法的:

```
(a > 5) + 2
```

其值为3。

这是C语言不同于其他语言之处。

关系运算符常用来组成关系表达式作为某些语句的条件,故称条件表达式。

关于等于运算符是由两个代数式中的等号组成的,有时容易写成一个等号与代数式中的等号相混。在C语言中,一个等号是赋值运算符,它与等于运算符截然不同,请一定注意其区别。

3.1.4 逻辑运算符

逻辑运算符是用来对操作数进行逻辑操作的。C语言提供了如下的逻辑运算符。

单目的逻辑运算符: !表示逻辑求反或逻辑非。如, $!(a + b)$ 。

双目的逻辑运算符: && 表示逻辑与,即对两个操作数进行逻辑求与。例如, $a \&\&b$ 。

|| 表示逻辑或,即对两个操作数进行逻辑求或。

关于逻辑求反是对真求反后为假,对假求反后为真。

关于逻辑求与是指两个操作数中只有都是真时,求与后才是真。否则,求与后为假。

关于逻辑求或是指两个操作数中只要有一个为真时,求或后就为真。只有两个都是假时,求或后才是假。

操作数的真与假是这样规定的:非零为真,零为假。

逻辑运算结果的真与假是这样规定的:真用1表示,假用0表示。

例如,

```
int a=5, b=0;
```

!a 的值为0,因为 a 为真,则 $!a$ 为假。

$a \& \& b$ 的值为0, 因为 b 为假, 则 $a \& \& b$ 为假。

$a || b$ 的值为1, 因为 a 为真, 则 $a || b$ 为真。

$!a || b$ 的值为0, 因为 $!a$ 和 b 都为假, 则 $!a || b$ 为假。

3.1.5 位操作运算符

位操作运算符又分为逻辑位运算符和移位运算符两类。

1. 逻辑位运算符

逻辑位运算符是一种对操作数按其二进制位进行逻辑操作的运算符。它包含如下几种:

单目运算符: \sim 表示按位求反运算符。如 ~ 5 。

双目运算符: $\&$ 表示按位与运算符。如, $5 \& 3$ 。

$|$ 表示按位或运算符。如, $3 | 5$ 。

\wedge 表示按位异或运算符。如, $5 \wedge 7$ 。

进行逻辑位操作时, 先将操作数化为二进制数, 然后按下列运算规则进行。

按位求反是将操作数中各个二进制位逐位求反, 即1求反后为0, 0求反后为1。

按位与是将两个操作数各二进制位从低位到高位对齐, 再将每位的两个二进制数相与, 除了两个“1”为1外, 其余为0。

按位或是将两个操作数各二进制位从低位到高位对齐, 再将每位的两个二进制数相或, 除了两个“0”为0外, 其余为1。

按位异或又称按位加, 忽略进位, 先将两个操作数各二进制位从低位到高位对齐, 再将每位的两个二进制数相加, 不考虑进位, 即两位相同的二进制数, 则为0, 两位是不同的二进制数, 则为1。

例如, 假定仅按8位二进制长度计算, 且不考虑符号位:

~ 5 为242

$5 \& 3$ 为1

$5 | 3$ 为7

$5 \wedge 3$ 为6

逻辑位运算符组成的表达式的值是算术值, 它与逻辑表达式不同。

2. 移位运算符

移位运算符是用来将某个操作数向某个方向(或左, 或右)移动所指定的二进制位数。移位运算符是双目运算符, 它包含有左移和右移两种。

$>>$ 表示右移运算符。如, $a >> 2$ 。

$<<$ 表示左移运算符。如, $b << 3$ 。

移位运算符是对某个操作数进位移位操作, 由于所移动是二进制位数, 需将待移位的操作数化为二进制数, 然后按指定的移动位数或向右移动或向左移动。

右移运算时, 移去的位被弃掉, 左端补0或补符号位。根据机器不同而定, 有的机器是补0, 有的机器是补符号位, 所谓符号位是指机器所存放的一个字的最高位。

左移运算时, 移去的位被丢掉, 右端一律补0。

例如,

$4 >> 2$ 的值为1。

4<<2的值为16。

移位运算符组成的表达式的值也是算术值。可以通过下述例子,上机试一下该机器右移位运算时是补0还是补符号位。

计算表达式 $-1>>2$ 的值,如果是-1,则右移运算时补符号位,否则是补0。

3.1.6 赋值运算符

赋值运算符是用来给变量赋值的。它是双目运算符,用来将一个表达式的值送给一个变量。C语言中,赋值运算符有一个基本的和10个复合的。

基本的赋值运算符是 $=$ 。该运算符左边一般是变量或表示某个地址的表达式,称为左值,该运算符的右边一般是一个表达式,称为右值。

复合的赋值运算符包含由5个算术运算符与基本赋值运算符组成的和由5个位操作运算符与基本赋值运算符组成的,它们分别是:

$+=$ 表示加赋值运算符。如, $a+=b$ 等价于 $a=a+b$ 。

$-=$ 表示减赋值运算符。如, $a-=b$ 等价于 $a=a-b$ 。

$*=$ 表示乘赋值运算符。如, $a*=b$ 等价于 $a=a*b$ 。

$/=$ 表示除赋值运算符。如, $a/=b$ 等价于 $a=a/b$ 。

$\%=$ 表示取余赋值运算符。如, $a\%=b$ 等价于 $a=a\%b$ 。

$\&=$ 表示位与赋值运算符。如, $a\&=b$ 等价于 $a=a\&b$ 。

$|=$ 表示位或赋值运算符。如, $a|=b$ 等价于 $a=a|b$ 。

$\wedge=$ 表示位异或赋值运算符。如, $a\wedge=b$ 等价于 $a=a\wedge b$ 。

$>>=$ 表示右移赋值运算符。如, $a>>=b$ 等价于 $a=a>>b$ 。

$<<=$ 表示左移赋值运算符。如, $a<<=b$ 等价于 $a=a<<b$ 。

复合赋值运算符所表示的表达式不仅比一般赋值运算符表示的表达式简练,而且所生成的目标代码也较少。因此,C语言程序中尽量采用复合赋值运算符的形式表示。

赋值运算符是一种有副作用的运算符。该运算符可产生一个表达式的值,同时又将改变其变量的值。例如,

```
x=5;
```

它将表示如下两层意思:一是使变量的值被改变为5,不论变量x原值是多少,执行上述语句后,x的值为5。二是表达式 $x=5$ 的值也是5,可将该表达式的值赋给另一个变量,也可以将该表达式的值输出显示。因此,下面的语句是正确的,并能验证 $x=5$ 表达式的值是5。

```
int x=3, m;  
printf("%d\n", x=5);  
m=x=8;  
printf("%d\n", m);
```

第一个printf()语句输出为5,它表明表达式 $x=5$ 的值为5。这时如果输出x的值也应该是5。第二个printf()语句输出为8,它表明m的值是8,而m的值是通过表达式 $x=8$ 的值获得的。因此,在C语言的程序中,连续赋值是有意义的。表达式 $m=x=8$;是先将8赋值给x,再将 $x=8$ 表达式的值8赋给m。所以,m获值为8。在使用赋值运算符时应注意该运算符的副作用。

3.1.7 其他运算符

除了上述的6类运算符外,所有其他运算符在这里讲述。

1. 三目运算符

三目运算符是由两个字符组成,要求三个操作数的一种特殊运算符,该运算符的功能类似于条件语句,故称条件运算符。其格式如下所示:

`d1? d2;d3`

其中,d1,d2和d3是三个表达式。?:是三目运算符。该运算符的功能是:先计算d1,如果d1的值是非0,则整个表达式的值是d2的值;如果d1的值是0,则整个表达式的值是d3的值。关于整个表达式的类型将是d2和d3这两个表达式中类型高的一个。

2. 逗号运算符

在C语言中,逗号(,)既可作分隔符又可作运算符。逗号运算符是用来把若干个表达式连接起来合成一个表达式,逗号表达式的值则是组成它的最后一个表达式的值,它的类型也是最后一个表达式的类型,逗号运算符较少使用,它只是用在只允许出现一个表达式的地方而又要同时出现几个表达式时,用它将几个表达式连起来组成一个逗号表达式。

3. 强制类型运算符

该运算符是用来强制一个表达式的类型的,该运算符是由一对圆括号括起一个类型说明符组成的,它作用于某个表达式的前面,便将这个表达式的类型强制成圆括号中所指定的类型。例如,

`(int)(a * b + c * 1.5)`

将表达式 `a * b + c * 1.5` 的类型强制成为 int 型。这种强制是暂时的,它不改变表达式中各变量原来的类型。对于同一个表达式可根据需要强制成这种类型或那种类型,不论表达式的类型比强制成的类型高还是低,都是可以的。

强制类型运算符是单目运算符,在程序中经常使用。

4. sizeof 运算符

该运算符用来求得某种类型或某个变量所占内存的字节数。它是一个单目运算符,作用于类型说明符或变量名的左边,并用圆括号将作用的类型说明符或变量名括起来。其格式如下所示:

`sizeof(<类型说明符>或<变量名>)`

例如:

`int a, b[10];`

使用 `sizeof(a)` 将获得变量 a 在机器内存中所占的字节数。一般的16位微机,表达式 `sizeof(a)` 的值应该为2,即与 `sizeof(int)` 的值是等价的;同样, `sizeof(b)` 的值为20,它是数组 b 的所有元素所占的总内存字节数。

通过下述语句便可求得你所使用的机器不同类型变量所占的内存字节数:

`printf("%d, %d, %d, %d, \n", sizeof(char), sizeof(int), sizeof(float), sizeof(double));`

该语句输出显示的结果将告诉我们,char 型、int 型、float 型和 double 型变量所占内存的字节数。如果你所使用的是一般的16位微机,其结果为1,2,4,8。

5. 元素/成员运算符

C 语言中提供了三个表示数组元素和结构/联合成员的运算符。

(1) 下标运算符:[]是用来表示数组元素的。[]内将给出标识该数组元素位置的下标表达式,该表达式是一个常量表达式。

(2) 结构/联合变量成员运算符:·是用来表示结构/联合变量的成员的。详见本书“结构联合枚举”一章。

(3) 指向结构/联合变量指针成员运算符:→是用来表示指向结构/联合变量的指针成员的。详见本书“结构联合枚举”一章。

6. 取地址运算符

该运算符是用来获取某个变量的地址值的,它是一个单目运算符。其格式如下所示:

&<变量名>

其中,&是取地址运算符。该运算符可作用在变量名、数组元素名、结构变量名前,不能作用在常量、表达式和数组名前。因为常量和表达式是没有内存地址的,数组名本身已是一个地址值了。例如,

```
int a, b;
```

&a 表示取变量 a 的地址值,即是变量 a 在内存中被分配的内存地址值,只有被存放在内存中的变量才可取地址值,同样,&b 表示取变量 b 的内存地址值。

7. 取内容运算符

该运算符是用来间接地获取某变量的值。假定,一个指针 p 指向某个变量 a,并且变量 a 在内存中存放的值为 5。将该运算符作用于 p(即 *p)则表示取 p 的内容,而 p 的内容是它所指向的变量 a 的值。因此,取内容运算符是用来获取被该运算符所作用的指针所指向的变量的值。该运算符实际上是通过指针来间接取变量的值。该运算符是一个单目运算符,运算符的符号是 *。例如,

```
int a=5, *p=&a;
```

*p 的值是 5。即指针 p 的内容是 5,也就是指针 p 所指向的变量 a 的值。

关于取内容运算符和前面的取地址运算符将在本书“指针”一章中还会详述。

8. 括号运算符

该运算符是用来改变原来的优先级的,括号运算符的优先级最高。括号运算符可以包含使用(即嵌套),即在括号内还可以使用括号,在出现多重括号时,应该先作最内层括号,按从里向外的顺序进行。在实际编程中,经常使用括号来改变优先级,即在某些表达式中要求先作优先级低的运算符,这时只好用括号来改变优先级。例如,在下述表达式中,

```
c=getchar()!=EOF
```

其中,c 是一个 int 型变量,可用来存放字符。getchar()是前面曾介绍过的从键盘上获得一个字符的函数,EOF 是一个符号常量,其值被定义为 -1。该表达式要求从键盘上获取一个字符后先放在变量 c 中保存,然后再判断该字符是否不等于 -1。一般的字符是不会等于 -1 的。作为终止键盘输入的符号(<ctrl>+z+回车)才是 -1。而上述表达式按运算符的优先级是先将获取的字符判断是否 -1,然后将判断结果赋给变量 c。显然,上述表达式与要求不一致,为此加括号改变优先级后将与要求相符:

```
(c=getchar())!=EOF
```

括号在 C 语言程序中是经常使用的,也是十分重要的。

以上共讲述了44个运算符的功能及使用时应注意的事项。

3.2 运算符的优先级和结合性

前节介绍了 C 语言中运算符的功能,本节进一步讨论运算符的优先级和结合性。优先级和结合性都是解决表达式计算顺序的重要依据,因此,计算表达式之前,一定要搞清组成该表达式的各运算符的优先级和结合性。

3.2.1 运算符的优先级

C 语言中运算符的优先级共分15级。记住每种运算符的优先级在计算表达式的值或书写表达式语句中是十分重要的。下面讲述如何记住优先级的方法。

记优先级妙诀如下:

去掉一个最高级,再去掉一个最低级。

一、二、三和赋值。

说明:最高级是括号和元素及成员(共4个)。最低级是逗号(1个)。一是单目运算符(共9个),二是双目运算符(共18个),三是三目运算符(1个),赋值是赋值运算符(11个)。接着,在双目运算符中,还包含有10个优先级。这10个优先级记忆方法是:

算术2,关系2,逻辑2;移位1插在前,逻辑位3插在后。

说明:“算术2”表明算术运算符又分二个优先级,*、/和%在前,+、-在后。“关系2”表明它在算术运算符后边有二类优先级,<、<=、>、>=在前,! =、==在后。“逻辑2”表明它在关系运算符之后,又分2个优先级,&&在前,||在后。“移位1插在前”表明移位运算符是1个优先级插在算术和关系之间,即>>和<<。“逻辑位3插在后”表明逻辑位运算符有3个优先级,&在前,^在中,|在后,它们插在关系和逻辑之间。这样,15种优先级的顺序就记住了。

3.2.2 运算符的结合性

前面讲述了运算符的15种优先级,优先级高的先作运算,优先级低的后作运算。那么,有若干个优先级相同的运算符组成的表达式,先作哪个呢?这就要看结合性了。结合性分两类,一类是从左至右,一般运算符都属于一类;另一类是从右至左,只有少数的3类运算符属于这一类,它们是:单目、三目和赋值。可见,只要记住了三类从右至左结合性的运算符就可以了,因为其余都是从左至右。

优先级和结合性虽然都是用来判断表达式的计算顺序的,但是它们却是两个完全不同的概念,优先级有15类,结合性只有2种。判断表达式计算顺序时,先按优先级高的先计算,优先级低的后计算,当优先级相同时再按结合性,或从左至右顺序计算,或从右至左顺序计算。例如,

$a - b * c + d;$

这个表达式计算顺序,先按优先级应先作 $b * c$,然后,剩下的运算符“-”和“+”的优先级相同,再按自左至右的结合性顺序,先作减法运算,后作加法运算。从这里可以进一步看出记忆优先级和结合性对解决计算顺序问题是何等的重要!

下面给出一张表,该表中清楚标明 C 语言中运算符的种类、名称、优先级和结合性。

优先级	类 别	运算符	名 称	结合性
0		() [] . ->	括号 下标 箭头 点	从左至右
1	单目	! ~ ++ -- - ((<类型>)) * & sizeof	逻辑反 按位反 增1 减1 求负值 强制类型 取内容 取地址 字节数	从右至左
3	算术	* / %	乘 除 取余	从左至右
4		+ -	加 减	
5	移位	<< >>	左移 右移	
6	关 系	> >= < <=	大于 大于等于 小于 小于等于	
7	逻辑位	== !=	等于 不等于	
8		&	按位与	
9	逻辑	^	按位异或	
10			按位异或	
11		&&	逻辑与	
12			逻辑或	
13	条件	?:	条件	从右至左
14	赋值	= +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=,	赋值 算术复合赋值 按位复合赋值	
15	逗号	,	逗号	从左至右

3.3 表达式

3.3.1 表达式和表达式的种类

1. 表达式

表达式是常量、变量、函数和运算符组合起来的有意义的式子。

下面都是合法的表达式：

```
++a-3
b/=4
x<=y
(m=1)&& n*6
a=1, b=2, c=3
x+sin(y)
x<2?m:n
```

合法表达式经过计算后都将具有一个确定的值和类型。

下面是一些非法表达式：

```
5.3%2
++3+2
++-a-b
```

非法表达式是没有值和类型的，在编译中系统将指出表达式是非法的。

C 语言是一个多表达式语言，这意味着 C 语言中有种类繁多的表达式，它包含了别的语言中所没有的一些表达式，在 C 语言程序中表达式语句占的比例很大，关于表达式语句下一章讲述。

2. 表达式的种类

这里介绍 C 语言中的各种表达式。表达式的种类与运算符的种类有关，由于 C 语言中运算符种类多，因此，表达式种类也多。它们是算术表达式、关系表达式、逻辑表达式、赋值表达式、条件表达式和逗号表达式。

(1) 算术表达式

由算术运算符、增1减1运算符和位操作运算符组成的表达式都属于算术表达式。

算术表达式的值为整型的称为整型表达式，其值为浮点型的称为浮点型表达式。算术表达式的类型由参与算术表达式的各操作数的类型决定的。如果参与算术表达式的各操作数的类型都相同，则表达式的类型与操作数的类型相同，如果参与算术表达式的各操作数的类型不同，则表达式的类型由操作数中类型最高的决定。

下面通过例题来熟悉和掌握各种算术表达式的计算方法。

[例3.1] 整型数运算。

```
main()
{
    int x,y;
    printf("Input x,y: ");
    scanf("%d%d",&x,&y);
```

```

printf("x+y=%d,x-y=%d\n",x+y,x-y);
printf("x*y=%d,x/y=%d\n",x*y,x/y);
printf("(x/y)*y+x(mod)y=%d\n",x/y*y+x%y);
}

```

执行该程序后,屏幕上显示如下提示信息:

Input x, y: 12 5

键入12,空格,5回车后,屏幕上显示如下输出信息:

```

x+y=17, x-y=7
x*y=60, x/y=2
(x/y)*y+x(mod)y=12

```

这里,需要说明的是两个整数相除,结果为整数,即 x/y 结果为2。而 $x\%y$ 的结果也为2,前者是整数商为2,后者是相除后余数为2。本例中出现的表达式都是整型的算术表达式。

[例3.2] 浮点数运算。

```

main()
{
    float a,b=8.5;
    a=7.2;
    a*=b;
    printf("%f\n",a);
    a/=b;
    printf("%.4f\n",a);
    printf("%.10f\n",a);
}

```

执行该程序的输出显示结果如下:

```

61.199997
7.2000
7.1999998093

```

本例中出现的表达式 $a*=b$ 和 $a/=b$ 都是浮点型算术表达式。对 float 型结果可按 %f 输出,也可按 %lf 输出,两者结果可能有不同,这是由机器表示数的精度造成的。

[例3.3] 增1减1运算。

```

main()
{
    int a,b,c;
    a=b=c=5;
    a=++b-++c;
    printf("%d,%d,%d\n",a,b,c);
    a=b+++c++;
    printf("%d,%d,%d\n",a,b,c);
    a=++b+c++;
    printf("%d,%d,%d\n",a,b,c);
    a=b--+--c;
    printf("%d,%d,%d\n",a,b,c);
}

```

```

a=b---c;
printf("%d,%d,%d\n",a,b,c);
a=-b++-c;
printf("%d,%d,%d\n",a,b,c);
}

```

本例题专门用来练习增1减1运算符的使用方法的。先说明一下关于在表达式中连续出现运算符时的写法问题。在C语言程序中有些表达式会连续出现两个或多个运算符,应该在多个运算符之间用空格符分隔。例如

```
a=b++++c++;
```

其中,++与+之间用空格符分隔,这样比较清晰,准确。但是,有时多个运算符之间也可以不加空格,这时系统将按“尽量取大”的原则来自动分隔运算符。例如:

```
a=b---c;
```

按“尽量取大”的原则,将该表达式连续出现的运算符分隔为:

```
a=b--_+_c;
```

因为b后边可跟一个“-”表示减去,也可跟一个“--”表示减1,到底用哪个,按“尽量取大”,便采用了b--,然后再减c。如果要写下述表达式,

```
a=b-_+_--c;
```

则不可写成

```
a=b---c;
```

否则,将被认为是

```
a=b--_+_c;
```

下面来分析该程序的输出结果:

已知,a,b,c值都为5。

执行表达式a=++b-++c后,b的值改为6,c的值改为6,++b值为6,++c值为6,所以,a值为0。第一个printf()语句输出为0,6,6。

执行表达式a=b++++c++后,b和c值都为7,b++值为6,c++值为6,所以a值为12。第二个printf()语句输出为12,7,7。

执行表达式a=++b+c++后,b和c的值都为8,++b值为8,++c值为7,所以,a值为15,第三个printf()语句输出为15,8,8。

执行表达式a=b--+-c后,b和c的值都为7,b--值为8,--c值为7,所以,a值为15,第四个printf()语句输出为15,7,7。

执行表达式a=b--_+_c后,b值为6,c值不变,b--值为7,所以a值为0。第五个printf()语句输出为0,6,7。

执行表达式a=-b++++c后,b值为7,c值不变,-b++值为-6。所以,a值为1。第六个printf()语句输出为1,7,7。

[例3.4]逻辑位运算。

```

main()
{
    unsigned a,b,c;
    a=0164;

```



```

    b=0xab;
    printf("a10=%d,a8=%o,a16=%xn",a,a,a);
    c=a|b;
    printf(" %d,%o,%x\n",c,c,c);
    c=a&b;
    printf(" %d,%o,%x\n",c,c,c);
    c=a^b;
    printf(" %d,%o,%x\n",c,c,c);
    c=~a+~b;
    printf(" %d,%o,%x\n",c,c,c);
}

```

执行该程序输出结果如下:

```

a10=116, a8=164, a16=74
255, 377, ff
32, 40, 20
223, 337, df
-289, 177337, fedf

```

说明:先将 a 和 b 的值在16位机上用二进制表示,a 的二进制表示为0000000001110100,b 的二进制表示为0000000010101011.因此,a|b 为0000000011111111,十进制表示为255.a&b 为000000000100000,十进制表示为32.a^b 为0000000011011111,十进制表示为223.~a 为111111110001011,~b 为1111111101010100.~a+~b 为1111111011011111,这是个负数的补码表示,化为原码为-0000000100100001,十进制表示-289.所以,本例将得上述结果。

[例3.5] 移位运算。

```

main()
{
    unsigned a=0x3d;
    a>>3;
    printf(" %x\n",a);
    a<<3;
    printf(" %x\n",a);
    a>>=4;
    printf(" %x\n",a);
    a<<=4;
    printf(" %x\n",a);
}

```

执行该程序后,输出如下结果:

```

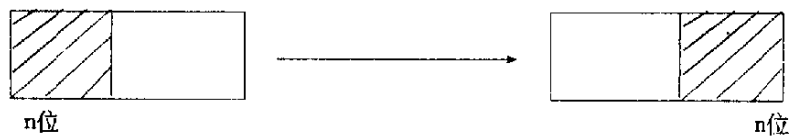
3d
3d
3
30

```

说明:移位运算是一种二进制位运算,该运算不改变其变量值。在表达式 a>>3 中,该表达式的值为十六进制的7,但是 a 仍为十六进制的3d,因此,输出 a 的按 %x 值仍为3d。同样,表达式 a<<3 的值为十六进制的1e8,但是 a 仍为十六进制的3d,因此,输出 a 的按 %x 的值仍为

3d。表达式 $a \gg 4$ 是一个赋值表达式它是将 $a \gg 4$ 表达式的值赋给 a 即改变了 a 的值为3。因此,按 $\%x$ 输出 a 的值为3。表达式 $a \ll 4$ 也是一个赋值表达式,它是将 $a \ll 4$ 的值赋给 a ,即改变了 a 的值为十六进制30。因此,按 $\%x$ 输出 a 的值为30。

[例3.6] 编程实现循环左移 n 位。如下图所示:



所谓循环左移是指将左移时丢掉位补在右边。以8位二进制为例,将10110001循环左移3位后,变为10001101。

实现循环左移几位程序如下:

```
main()
{
    unsigned x,y,z;
    int n;
    printf("Input x,n: ");
    scanf("%o%d",&x,&n);
    y=x>>(16-n);
    z=x<<n;
    z|=y;
    printf("%o\n",z);
}
```

执行该程序后,屏幕显示如下信息:

Input x, n: 1365354

输出如下结果:

15273

说明:所键入的 x 值的二进制表示为:1011110101011101,将它左移4位,并将移掉的位补到右边,则变为1101010111011011,即八进制数表示为1 5 2 7 3 3。

该程序的编程步骤如下:

被移的数字放在 x 变量中,移位后输出的数字放在 z 变量中,用 y 变量保存一下中间结果。

① 先将 x 中被左移 n 位后移出的位放到 y 变量中,为此,作如下操作:

$y=x \gg (16-n);$

② 将 x 变量左移 n 位,低位补0,并放到变量 z 中,为此,作如下操作:

$z=x \ll n;$

③ 再将 z 与 y 相按位或,并放到变量 z 中,则变量 z 中内容为输出结果。

$z|=y;$

[例3.7] 位操作运算。

```
main()
{
    int a=15,b=7;
```

```

a/=2;
b%=2;
printf("%5d%5d%5d%5d\n",a&b,a|b,a^b,~b);
printf("%5d%5d\n",a>>1,b<<2);
}

```

执行该程序输出结果如下：

```

1    7    6   -2
3    4

```

(2) 关系表达式

由关系运算符组成的表达式称为关系表达式。关系表达式的值为0或1，其类型为 int 型。当某种关系成立时，即为“真”时，其值为1；否则其值为0。数值比较按其值的大小进行比较，字符比较按其字符的 ASCII 码值的大小，即在字符表中排列顺序进行。例如，

'm' < 'n'; 其值为1，因为此式成立。
 126 < -5; 其值为0，因为此式不成立。
 'a' == 'A'; 其值为0，因为此式不成立。
 关系表达式常常被用作语句的条件。

[例3.8] 求下列各关系表达式的值。

已知：char c='m';
 int a=1, b=3, c=-5;
 double x=7e+33, y=0.0001;

求下列各表达式的值：

- ① 'a'+3<c
- ② 7<b<3
- ③ -a-3*b>=c-1
- ④ x-2.7<=x+y
- ⑤ x<x+y
- ⑥ 'w'==c+9
- ⑦ a+b-c==2*-b
- ⑧ c==b-5==a
- ⑨ -a+b>=0
- ⑩ x!=x+y

分析计算后各表达式的值为：

- ① 1 ② 1 ③ 0 ④ 1 ⑤ 0
- ⑥ 1 ⑦ 0 ⑧ 0 ⑨ 1 ⑩ 0

说明：

① 在一个很大很大的数(如, 7e+33)与一个与它相比很小很小的数进行加减运算时, 其结果仍为那个很大很大的数。根据这种关系可判如④题为1, ⑤题为0和⑩题为0。

② 计算表达式7<b<3时, 由于运算符优先级相同, 故按其结合性从左至右顺序计算, 因为7<b 为0, 而0<3为1。所以, 该表达式值为1。

③ 计算表达式 $a+b-c==2*-b$ 时,按优先级应将 $==$ 两边的表达式 $a+b-c$ 和 $2*-b$ 的值计算出来,再判断它们是否相等,因为 $a+b-c$ 的值为9,而 $2*-b$ 的值为6,所以该表达式的值为0。

其他表达式结果不再一一解释。请读者分析,且上机验证一下。

(3) 逻辑表达式

逻辑表达式是由逻辑运算符组成的表达式,逻辑表达式的值在 C 语言中用1或0表示,其类型为 int 型。C 语言中,关于逻辑表达式的运算规定如下:

非零值为真,真用1表示;零值为假,假用0表示。

例如, `int x=-56;` 求表达式 `!x` 的值和 `x&& x-2` 的值。

因为非零值为真,即 `x` 为真, `!x` 则为假,假用0表示,所以表达式 `!x` 的值为0。同样, `x` 值为真, `x-2` 的值也为真, `x&& x-2` 的值应为真,真用1表示,所以表达式 `x&& x-2` 的值为1。

[例3.9] 求下列逻辑表达式的值。

已知: `char c='m';`

`int a=2, b=2, c=2;`

`double x=0.0, y=3.2;`

求下列表达式的值:

① `!x * !y`

② `!!y`

③ `x || a&& b-2`

④ `a < b && x < y`

⑤ `a < b || x < y`

⑥ `a == b && x <= y`

⑦ `x != y && b+1 == !c+3`

⑧ `'A' <= c && c <= 'z'`

⑨ `c-1 == 'l' || c+1 == 'n'`

⑩ `a == 2 || b == 3 || (c == 3)`

分析计算上述表达式的值如下:

① 0 ② 0 ③ 0 ④ 0 ⑤ 1

⑥ 1 ⑦ 1 ⑧ 1 ⑨ 1 ⑩ 1

说明:求逻辑表达式的值要按照上述的规则,先判断操作数是真还是假,再求得逻辑运算的值,最后按照真用1表示,假用0表示的规则得出结果。以本例中⑦题为例,它是求两个表达式 `x != y` 和 `b+1 == !c+3` 相与的结果。由于 `x != y` 为真,又 `b+1` 为3, `!c+3` 也是3,因此,表达式 `b+1 == !c+3` 的值为真,所以,整个表达式的值为真,其表达式值为1。

(4) 赋值表达式

赋值表达式是由赋值运算符组成的,赋值运算符包含基本赋值运算符和复合赋值运算符。赋值表达式在 C 语言程序中使用较多。

前面讲过了赋值运算符的特点,这里再强调一次,赋值运算符具有下述两大特点:

一是具有副作用;二是结合性从右向左。由于这两个特点,应该这样理解赋值表达式。

● 赋值表达式是一个运算表达式,它将赋值运算符右边的表达式的值传送给赋值运算符

左边的变量,从而改变其变量的值。与此同时,赋值表达式本身仍然有一个确定的值。实际上,它具有两个作用:一是改变变量值,二是表达式本身的值。

● 由于赋值表达式的两个作用,再加上它具有自右至左的结合性,因此,C语言中允许连续赋值。例如,

```
int a, b, c, d;  
a=b=c=d=1;
```

它等价于

```
a=1; b=1; c=1; d=1;
```

这种连赋值的方式不仅可以使得书写简单,而且可以提高执行速度。

● 在赋值表达式中,右值和左值的类型如果不一致时,则右值类型自动转换成左值类型。右值一般是表达式,左值是变量名,也可以是表达式,该表达式要是用来表示内存地址值的,在“指针”一章中会看到。

[例3.10] 赋值运算。

```
main()  
{  
    int a,b,c;  
    printf("%d,%d\n",a=4,b=5);  
    c=(a=6)+(b=9);  
    printf("%d\n",c);  
    a=b=c=a+3;  
    printf("%d,%d,%d\n",a,b,c);  
    a+=b-=c/=a-2;  
    printf("%d,%d,%d\n",a,b,c);  
}
```

执行该程序输出结果如下:

```
4,5  
15  
9,9,9  
17,8,1
```

说明:第一个 printf()是输出两个赋值表达式 a=4和 b=5的值,分别为4和5。

第二个 printf()是输出 c 的值,即表达式(a=6)+(b=9)的值,显然是15。

第三个 printf()是输出 a, b, c 的值,在执行 a=b=c=a+3表达式后,由于 a 值为6, a+3 为9,即将9连赋值给 a, b, c 所以,输出结果为9,9,9。

第四个 printf()也是输出 a, b, c 的值。在执行 a+=b-=c/=a-2表达式时,按结合性从右自左,先计算 c/=a-2的值,由于 a 值为9, a-2值为7, c=c/7,即 c 值为1;再计算 b-=1 的值,由于 b 值为9,即 b=b-1, b 值仍为8。最后计算 a+=8的值,由于 a 值为9, a=a+8, 所以, a 值改变为17。整个表达式值为17。

(5) 条件表达式

由三目运算符组成的表达式称为条件表达式,该表达式具有条件语句(if-else 语句)的功能。

[例3.11] 条件表达式运算。

```

main()
{
    int a,b,c;
    a=1; b=2; c=3;
    a+=b+=c;
    printf("%d\n",a<b?b:a);
    printf("%d\n",a<b?a++,b++);
    printf("%d,%d\n",a,b);
    printf("%d\n",c+=a>b?a++:b++);
    printf("%d,%d\n",b,c);
    a=3; b=c=4;
    printf("%d\n",(c>=b&&b==a)?1:0);
    printf("%d\n",c>=b&&b>=a);
}

```

执行该程序输出结果如下：

```

6
5
6,6
9
7,9
0
1

```

说明：条件表达式的功能相当于一个 if 语句。在计算条件表达式的值时，先找出“条件”来，然后计算其值分析是非零还是零来选取冒号(:)前还是冒号后的表达式的值作为条件表达式值。“条件”是在问号(?)前的表达式，确定该表达式是关键。例如，本例中有下述表达式，

$c+=a>b?a++:b++$

如果认为 $c+=a>b$ 是条件表达式中的“条件”，则是错误的。仔细分析上述表达式可以看出，这个表达式是一个复合赋值表达式，即将条件表达式的值与 c 相加后再赋值给 c。因此，条件表达式中的“条件”是 $a>b$ 。按上面给出 a 和 b 的都是 6，则 $a>b$ 为假，即 0，所以条件表达式的值为 $b++$ 的值，即 6，而 c 的值为 3，因此，将 9 赋给 c，该表达式的值为 9。从上述分析可以看出优先级对于求表达式的值是十分重要的。这里，三目运算符的优先级高于赋值运算符。

(6) 逗号表达式

由逗号运算符将若干个独立的表达式连接起来组成一个表达式，该表达式被称为逗号表达式。逗号表达式的值和类型都是最后一个表达式的值和类型。逗号表达式的计算顺序是自左至右顺序进行。例如，

$a=5, b=7, c=a+b$

这是一个逗号表达式，执行该表达式时，先计算 $a=5$ ，a 值改变为 5，再计算 $b=7$ ，b 值改变为 7，最后计算 $c=a+b$ ，其值为 12，这将是整个逗号表达式的值。

[例3.12] 逗号表达式运算。

```

main()
{

```

```

int a,b,c;
c=(a=5,b=a*3);
printf("%d\n",c);
printf("%d\n",(a=2+6,a*2,a*5));
c+=(a=b=3,b=a*c,a+b);
printf("%d,%d,%d\n",a,b,c);
}

```

执行该程序输出结果如下：

```

15
40
3,45,63

```

说明：在有逗号运算符参与的表达式中，逗号运算符的优先级最低。例如，本例中下述表达式，

$a=2+b, a*2, a*5$

与表达式

$a=(2+6, a*2, a*5)$

是不同的。

前边是一个逗号表达式，其值是 $a*5$ 表达式的值；而后边是一个赋值表达式，将一个逗号表达式的值赋给 a 变量。

3.3.2 表达式的值和类型

任何一个表达式都具有一个确定的值和一种类型。正确书写表达式和计算表达式的值和类型是编程和分析程序中重要的工作。

在书写表达式和计算表达式时，必须搞清楚表达式的计算顺序。表达式的计算顺序首先是由运算符的优先级决定的，优先级高的先做优先级低的后做；其次是由运算符的结合性决定的，在优先级相同的情况下，由结合性决定，有少数运算符的结合性从右至左，而多数运算符的结合性是从左至右。例如，

$x=y<<4;$

该表达式有两个运算符， $=$ 和 $<<$ ，按优先级是 $<<$ 高于 $=$ ，因此，它等价于

$x=(y<<4);$

表明先作 $y<<4$ ，将其值再赋给变量 x 。

在书写表达式时，如果个别运算符的优先级忘记了，可以使用圆括号来改变优先级。多重括号时最内层的优先级最高。

表达式的类型将是参与该表达式的各种操作数中类型最高的。例如，

$\text{float } a=7.6;$

$a+'a'+7$ 的类型应该是 a 变量类型，即 float 型。字符 $'a'$ 转换 int 型，再转换为 float 型，整数 7 也转换为 float 型，结果为 float 型。

计算出下列表达式的值，并指出其类型。

$\text{int } a=5;$

$\text{float } m=5.5, n=3.2;$

$m+n-a\%3*(int)(m-n)\%2/2$

计算该表达式,按优先级先计算括号内的, $m-n$ 为 2.3,取 int 型数后为 2,即变为

$m+n-a\%3*2\%2/2$

再按优先级顺序计算 $a\%3*2\%2/2$ 的值, $a\%3$ 为 2, $2*2$ 为 4, $4\%2$ 为 0, $0/2$ 为 0,即变为

$m+n-0$

其值为 8.7,类型为 float 型。

在书写表达式时,为了清晰明了或与系统对连续出现的运算符缺省分隔不同,在遇到连续出现的运算符时,用空格符分隔开。例如,

$a++ ++b;$

$x++ ++y++;$

又例如,

$a---b;$

系统缺省的分隔法是“尽量取大”,于是被认为是

$a---b;$

如果原意是要求 a 减去 $---b$,则必须按下列写法:

$a---b;$ 或者

$a-(-b);$

另外,由于不同的运算符采用相同的符号,例如,连接符(—),既可作单目运算符的求负运算,又可作双目运算符的减法运算;星号符(*),既可作单目运算符的取内容运算,又可作双目运算符的乘法运算等等。因此,在计算表达式的值之前,应该先确定各运算符的功能,确认正确后再作求值运算。例如,

$!-10-9<1$

这个表达式中有 4 个运算符,经确认后,!是逻辑求反,第一个“—”是单目运算符求负,第二个“—”是双目运算符减法,<是条件运算符小于。确认后,按优先级判断计算顺序,先作单目运算符,由于两个单目运算符同时出现,则按结合性从右自左,即先对 10 求负,得 -10,再对 -10 求反,则为 0,再计算 $0-9$ 为 -9,最后计算 $-9<1$,则为 1,所以,上述表达式的值为 1。

3.3.3 表达式求值中值得注意的两个问题

前边讲述了表达式求值的方法,这里讲解表达式求值时两个值得注意的特殊问题。

(1) 在包含 && 和 || 运算符的逻辑表达式的求值过程中,当计算出某个操作数的值后就可以确定整个表达式的值时,计算便不再继续进行。这就是说,并不是所有的操作数都被求值,只是在必须求得下一个操作数的值才能求出逻辑表达式的值时,才计算该操作数的值。例如,在由一个或多个 && 运算符组成的逻辑表达式中,自左向右计算各个操作数时,当计算到某个操作数的值为零时,则不再继续进行计算,这时,该逻辑表达式的值为 0;同样,在由一个或多个 || 运算符组成的逻辑表达式中,自左向右顺序计算各个操作数时,当计算到某个操作数的值为非零时,则不再继续进行计算,这时该逻辑表达式的为 1。这就是说,在由 && 运算符组成的逻辑表达式中,只有所有的操作数都不为零时,所有的操作数才都被计算;而在由 || 运算符组成的逻辑表达式中,只有所有的操作数的值都不为非零时,所有的操作数才都被计算。这一点读者必须注意,下面通过例子加以说明。

[例3.13] 分析下列程序输出结果,注意程序中逻辑表达式的求值。

```
main()
{
    int x,y,z;
    x=y=z=1;
    ++x&&--y&&++z;
    printf("%d,%d,%d\n",x,y,z);
    --x||++y||++z;
    printf("%d,%d,%d\n",x,y,z);
    x=y=z=-1;
    ++x&&++y||++z;
    printf("%d,%d,%d\n",x,y,z);
    ++x||--y&&--z;
    printf("%d,%d,%d\n",x,y,z);
}
```

分析该程序的输出结果如下:

在计算表达式 $++x \&\&--y \&\&++z$ 之前, x, y, z 的值都是1。按顺序先计算 $++x$ 的值为2,接着计算 $--y$ 的值为0,则这时可判定该逻辑表达式的值为0,不再计算表达式 $++z$ 的值,因此,第一个 `printf()` 语句输出的结果为2,0,1。

在计算表达式 $--x || ++y || ++z$ 时,按顺序先计算 $--x$ 的值为1,则这时便可判定该逻辑表达式的值为1,不再计算 $++y$ 和 $++z$ 的值,因此,第二个 `printf()` 语句输出的结果为1,0,1。

在计算表达式 $++x \&\&++y || ++z$ 之前, x, y 和 z 的值都为-1,按顺序先计算 $++x$ 的值为0,可知 $++x \&\&++y$ 的值为0,则不必计算 $++y$ 的值,而接着计算 $++z$ 的值为0,这样该逻辑表达式的值为0。而第三个 `printf()` 语句输出结果为0,-1,0。

在计算表达式 $++x || --y \&\&--z$ 时,按顺序先计算 $++x$ 的值为1,这时可判知该表达式的值为1,不再去计算 $--y$ 和 $--z$ 的值。第四个 `printf()` 语句输出结果为1,-1,0。

(2) 在C语言中,编译系统可根据需要重新安排表达式和参数表的计算顺序。这样,在有副作用的运算符和参数表中,由于不同编译系统的计算顺序的不同,对同一个程序可能产生出不同的输出结果,这被称为二义性。出现二义性的原因在于以下两点:

- 表达式和参数表中有具有副作用的运算符,即增1减1运算符或赋值运算符;如果没有具有副作用的运算符,不会出现二义性。

- 不同的编译系统,并且有不同的计算顺序。对于同一个编译系统,或者计算顺序相同的不同的编译系统,也不会出现二义性。

在实际应用中,应该尽量避免由运算符的副作用引起的二义性,以便保证程序的兼容性。

下面通过几个例子说明具有副作用的运算符将会带来的二义性,并了解避免二义性的方法以及了解编译系统的计算顺序对分析程序输出结果的重要性。

[例3.14] 增1减1运算符组成的表达式引起的副作用。

```
main()
{
    int m,n;
    m=1;
```

```

    n = ++m + m;
    printf(" %d, %d\n", m, n);
}

```

分析该程序的输出结果如下：

由于不同编译系统可能会有不同的计算顺序，在计算 $++m + m$ 时，有的编译系统先求 $++m$ 的值再求 m 的值，然后求和，这时 n 的值为4， m 值为2，输出结果为2,4。而有的编译系统先求 m 值再求 $++m$ 的值，然后求和，这时 n 的值为3， m 的值为2，输出结果为2,3。由此可见，两种不同计算顺序的编译系统将对该程序有着两种不同的输出结果，这就是“二义性”。为了避免二义性的出现，可以将该程序作如下修改：

```

main()
{
    int m, n;
    m = 1;
    n = ++m;
    n = n + m;
    printf(" %d, %d\n", m, n);
}

```

任何不同的编译系统都将使该程序输出如下结果：

2,4

[例3.15] 赋值运算符组成的表达式引起的二义性。

```

main()
{
    int m, n;
    m = 1;
    n = (m = 7) + m;
    printf(" %d, %d\n", m, n);
}

```

分析该程序输出结果如下：

在计算表达式 $n = (m = 7) + m$ 时，有的编译系统先求 $m = 7$ 的值再求 m 的值，然后求和，其输出结果为7,14。有的编译系统先求 m 的值，再求 $m = 7$ 的值，然后相加，其输出结果为7,8。可见，赋值运算符的副作用也会造成不同编译系统的不同计算顺序所造成的二义性。为了避免这种二义性，该程序可进行如下修改：

```

main()
{
    int m = 1, n;
    m = n = 7;
    n = n + m;
    printf(" %d, %d\n", m, n);
}

```

修改后程序的输出结果为7,14。在任何的编译系统下都会输出上述结果。

[例3.16] 增1减1运算符在参数表中引起的二义性。

```
main()
{
    int i=1;
    printf("%d,%d\n",i++,i);
}
```

分析该程序输出结果:

在该例程序的 printf() 函数的参数表 i++, i 中, 有二个参数, 并且出现了具有副作用的运算符 ++。由于允许不同的编译系统有不同的计算顺序, 有的编译系统自左至右计算参数表中的参数, 即先计算 i++ 的值为 1, 这时 i 的值为 2, 再计算后一个参数 i 的值为 2, 则输出结果为 1, 2。有的编译系统自右至左计算参数表中的参数, 即先计算 i 的值为 1, 再计算 i++ 的值仍为 1, 则输出结果为 1, 1。可见不同编译系统由于计算参数表中参数顺序的不同, 会引起二义性。为避免二义性, 原程序可作如下修改:

```
main()
{
    int i=1, j;
    j=i++;
    printf("%d,%d\n", j, i);
}
```

该程序的输出结果如下:

1, 2

这个程序不再会引起二义性, 因为输出结果已经不再与参数表中参数的计算顺序有关。

练 习 题

1. C 语言中有哪些种类运算符? 为什么说 C 语言的运算符丰富、使用复杂?
2. 求余运算符 % 在使用时应注意些什么事项? $-5\%3$ 与 $5\%-3$ 的值相同吗?
3. 增1减1运算符有何特点? 它的副作用指的是什么? 使用时应注意哪些问题?
4. 关系运算符有哪些? 等于和不等运算符在使用时应注意些什么?
5. 逻辑运算符有哪些? 逻辑运算中逻辑值是如何规定的? 对操作数的真假又是如何规定的?
6. 位操作运算符的含意是什么? C 语言中有哪些位操作运算符?
7. 逻辑位运算符与逻辑运算符有何不同?
8. 逻辑位运算符中按位与运算符 & 和按位异或运算符 ^ 有何不同?
9. 移位运算符是双目运算符吗? 它的两个操作数如何规定?
10. 赋值运算符有哪些? 复合赋值运算符有何特点?
11. 为什么说赋值运算符具有副作用? C 语言中允许连续赋值吗? (即给相同值的变量一起赋值, 如, $a=b=c=5$) 为什么?
12. 三目运算符具有什么功能? 如何使用?
13. sizeof 运算符的功能是什么? 它可以作用于任何一种变量名上吗?
14. 单目运算符 & 和 * 有何作用?
15. 括号运算符的功能是什么? 它在使用中是否允许嵌套?
16. C 语言中运算符的优先级有哪些? 如何记忆不同种类运算符的优先级?

17. 什么是结合性?C 语言中有哪些种类的结合性?如何记忆各种运算符的结合性?
18. 如何计算表达式的值和确定表达式的类型?
19. 逻辑表达式求值时应注意什么问题?
20. 表达式和参数表的二义性是怎样出现的?如何避免二义性?

作 业 题

1. 判断下列描述是否正确,对者划√,错的划×。

- (1) 所有的算术运算符都可以对 int 型数或 float 型数进行操作。
- (2) 增1减1运算符前缀运算与后缀运算表达式的值都是一样的。
- (3) 增1减1运算符的前缀运算和后缀运算变量的改变值都是一样的。
- (4) C 语言规定,逻辑值真用非0表示,逻辑值假用0表示。
- (5) 逗号运算符的优先级最低。
- (6) 移位运算符中右移和左移时,一律用0补位。
- (7) sizeof 运算符可用来求得某种类型在内存中所占的字节数,也可用来求得某个已定义的变量在内存中所占的字节数。
- (8) 在一个优先级相同的运算符组成的表达式中,计算顺序将自左至右进行。
- (9) 一个表达式的类型是参与该表达式的操作数中类型最高操作数的类型。
- (10) 赋值表达式的值就是被赋值的变量的值。

2. 选择填空

(1) 下列各运算符中,优先级最高的是()。

A. ^ B. | C. & D. !

(2) 下列各运算符中,优先级最低的是()。

A. << B. <= C. != D. &&

(3) 下列表达式中,a 和 b 是已知 int 型变量,()是非法的。

A. a%(-5) B. a+=-1 C. b=a++ D. a=5*b=7

(4) 下列表达式中,已知 int a=3, b=10;()是合法的。

A. a+3.7%2 B. --(-b) C. --+(a+b) D. a+=b++++2

(5) 下列关于运算符的描述中,()是错误的。

A. 运算符的结合性有二种:一是自左到右,另一种是自右到左。

B. 仅有两类运算符有副作用:增1减1运算符和赋值运算符。

C. 所有的运算符既可对常量操作又可对变量操作。

D. 运算符的优先级和结合性是用来确定表达式计算顺序的。

3. 填空

(1) C 语言对于逻辑运算的规定是()为真,真用()表示,()为假,假用()表示。

(2) & 作为双目运算符表示(),作为单目运算符表示()。

(3) 增1减1运算符不可作用在()和()上。

(4) 三目运算符是(),用三目运算符组成的表达式称为()表达式,其值为(),其类型为()。

(5) 双目运算符中优先级由高到低分别是()、()和(),而移位运算符在()与()之间,逻辑位运算符在()与()之间。

4. 计算下列各表达式的值,并上机编程验证所计算的值是否正确。

(1) 计算下列各表达式的值,按%d输出。

- ① $15-8<<1+1$
- ② $4*5|2<<1$
- ③ $8==6<=4\&0xff$
- ④ $18\%5*3/7-2$
- ⑤ $5<=3+2-(8>3)$

(2) 已知, unsigned int a=0152, b=0xbb, 求下列各表达式的值,按%x输出。

- ① $a|b$
- ② $a\&b$
- ③ $a\^b$
- ④ $\sim a+\sim b$
- ⑤ $a<<=3$

(3) 已知, int i=10, j=2, 求下列各表达式的值按%d输出。

- ① $++i-j--$
- ② $i=i*=j$
- ③ $i=3/2*(j=3-2)$
- ④ $\sim i\^j$
- ⑤ $i\&j|1$

(4) 已知, int a=5, b=3, 求下列各表达式的值以及 a 和 b 的值,按%d输出。

- ① $!a\&\&b++$
- ② $a||b+2\&\&a*b$
- ③ $++a, b=10, a+5$
- ④ $a=1, b=2, a>b? ++a : ++b$
- ⑤ $a+=b\%=a+b$

第四章 语 句

本章集中讲述 C 语言的语句系列。C 语言提供了足够的语句,采用这些语句可以构成结构化程序设计的三种基本结构:顺序结构、选择结构和循环结构。语句是编程的基础,学好语句是编好程序的前提。本章将详细讲解 C 语言中各种语句的定义格式、功能和使用方法以及在使用中应注意的问题。学好这章的内容为后面的编程打下基础。

4.1 表达式语句和空语句

4.1.1 表达式语句

表达式语句是指任何一种表达式末尾加上分号(;)所组成的语句。在 C 语言程序中,表达式语句出现得最多,因此,有人说 C 语言是表达式语言。

表达式语句与表达式之间虽然只差一个分号(;),但二者是截然不同的。在程序中,有的地方需要用表达式,则不能写成表达式语句。例如,在 if 语句或循环语句的条件中,要求用表达式作为条件,如果写成表达式语句(即末尾加了分号),则是错误的。同样,在要求用表达式语句的地方,写成表达式(即少写一个分号),也是错误的。所以,读者一定要搞清楚何时用表达式,何时用表达式语句,二者不要搞混了。

例如,

`b=a+3`

是一个表达式;而

`b=a+3;`

便是一个表达式语句,这是一个赋值表达式语句。类似地,表达式语句还有:

<code>++i;</code>	(算术表达式语句)
<code>m>n? m:n;</code>	(条件表达式语句)
<code>b==5;</code>	(比较表达式语句)
<code>x&& ++y z;</code>	(逻辑表达式语句)
<code>printf("on!\n");</code>	(函数调用也是表达式语句)
<code>i=3, j=5, k=7;</code>	(逗号表达式语句)

等等。

有些表达式语句虽然是合法的,例如,

`m>n? m:n;`

但是,它并没有实际意义,需要将其表达式的值赋给某个变量才有意义。例如,

`x=m>n? m:n;`

这是一个有意义的表达式语句,它将一个表达式 `m>n? m:n` 的值赋给了变量 `x`。

虽然任何一个表达式加上分号都构成表达式语句,但在程序中要出现的是有意义的表达式语句。

4.1.2 空语句

空语句是一种只有分号而没有表达式的特殊语句。空语句是 C 语句中最简单的语句,因为它只由一个分号(;)组成。

空语句是一种不执行任何操作的语句。说它是特殊语句是指它是一种“不做事情”的语句。空语句在编程中也是有用的,它主要被用在需要一条不做事物的语句的地方。例如,它可用来作循环体,则该循环是空循环。下面是一个为了延迟一段时间的循环,其循环体可用空语句。

```
for(i=0; i<1000; i++)  
    ;
```

空语句还可以用在 goto 语句中语句标号的后面,即需要 goto 语句转向一条什么事情都不做的语句,那么在该语句标号的后面可用空语句。

4.2 复合语句和分程序

4.2.1 复合语句

复合语句是指由两条或两条以上语句用花括号({ })括起来的语句序列。

C 语句中的语句简单地划分为单条语句和复合语句两类。单条语句是指只有一条语句,而复合语句是指多条语句的总称,但是,多条语句用花括号括起来才称复合语句。没有用花括号括起的若干条单条语句只能称为语句序列。所以,复合语句是一种特殊的语句序列,它被一对花括号括起来,它在程序中被看作是一条语句。一般地说,凡是出现一条语句的地方都可以出现复合语句。

复合语句是 C 语言程序中常用的语句形式之一。在复合语句内部还可以包含有复合语句,即复合语句可以嵌套。例如,

```
    :  
    {  
        a=5;  
        :  
        {  
            b=7;  
            :  
        }  
        :  
    }  
    :  
    :
```

上述便是复合语句嵌套的一种形式。复合语句常用作 if 语句的 if 体、else 体和 else if 体以及循环语句的循环体等。

复合语句和函数体虽然都是用一对花括号({ })来定界的,但是两者间是有区别的。复合语句是由两条或两条以上语句序列组成的,而函数体内可以是多条语句的语句序列,也可以只有单条语句,还可以没有语句,即为空。另外,函数体内可以包含若干个复合语句,而复合语句内不能包含函数体,只能包含复合语句。

4.2.2 分程序

分程序又称程序块。分程序是一种花括号内含有说明语句的复合语句。分程序是一种复合语句,但是复合语句不一定是分程序,只有包含说明语句的复合语句才是分程序。因此,复合语句包含了分程序,分程序是复合语句的一种。具有下列格式的复合语句是分程序:

```
    <说明语句序列>
    <执行语句序列>
}
```

在分程序中,说明语句序列一定要放在执行语句序列的前边。

在分程序内可以定义或说明变量,也可以对已定义过的变量进行重新定义。在分程序中定义的变量都是属于局部变量,它的作用域仅在定义它的分程序内。有关这方面的详细情况在讲述存储类时再做详细描述。

4.3 分支语句

C 语言提供的分支语句有条件语句和开关语句两种,它们都可以用来实现多路分支。

4.3.1 条件语句

条件语句是一种根据指定的条件来决定执行不同的程序段的语句。条件语句又称 if 语句。

1. 条件语句的格式和功能

条件语句的格式如下:

```
if(<条件1>)
    <语句1>
else if (<条件2>)
    <语句2>
else if (<条件3>)
    <语句3>
    :
else if (<条件 N>)
    <语句 N>
else
    <语句 N+1>
```

这是 if 语句的完整的格式。其中,if 和 else 都是关键字。<条件1>至<条件 N>都是任一表达式,用来指出条件的表达式多是关系表达式和逻辑表达式,也可用其他表达式。<语句1>至<语句 N+1>都是单条语句或复合语句。

在 if 语句中,至少要有有一个 if 短语,最简单的 if 语句格式如下:

```
if(<条件>)
```


〈语句〉

在 if 语句中,除了至少要有有一个 if 短语之外,可以有一个(只能有一个)else 短语,或者没有 else 短语。else if 短语在 if 语句中可以没有,也可以有一个或多个。一般在多路分支的问题中,需要用到多个 else if 短语。具有一个 else 短语而无 else if 短语的 if 语句格式如下:

```
if(〈条件1〉)
```

```
    〈语句1〉
```

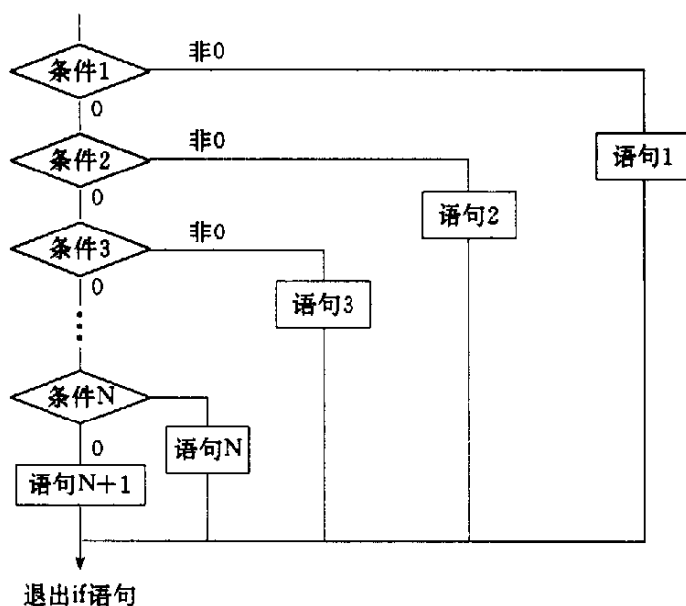
```
else
```

```
    〈语句2〉
```

if 语句的功能描述如下:

先计算〈条件1〉中表达式的值,如果其值为非零,则执行〈语句1〉,然后执行该 if 语句后边语句;如果其值为非零,则不执行〈语句1〉,接着计算〈条件2〉中表达式的值。如果其值为非零,则执行〈语句2〉,然后执行该 if 语句后边语句;如果其值为零,则不执行〈语句2〉,接着计算〈条件3〉中表达式的值,再判断,依次类推。如果 N 个条件对应的表达式的值全都为零,则执行〈语句 N+1〉,然后执行该 if 语句后边的语句。如果 N 个条件对应的表达式中某个值不为零,则执行该条件后边的语句,然后退出 if 语句,即执行该 if 语句后边的语句。

if 语句的功能可用下图所示:



2. if 语句的应用

if 语句在具体应用中要注意:一个 if 语句中只能有一个 else 短语,也可以没有 else 短语,而且在多重嵌套的 if 语句中,else 短语总是与它前面最近的 if 相匹配的。

下面通过几个具体例子说明 if 语句的使用。

[例4.1] 输入两个整数,按其值由大到小顺序输出这两个数。

```
main()
{
    int a,b,t;
```

```

printf("Input a,b: ");
scanf("%d%d",&a,&b);
if(a<b)
{
    t=a;
    a=b;
    b=t;
}
printf("%d,%d\n",a,b);
}

```

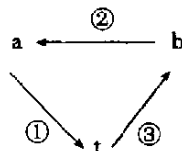
执行该程序后,屏幕显示如下提示信息:

Input a, b: 3 5 ✓

输入3和5后,按回车键。3与5之间用空格符分隔,屏幕显示如下结果:

5,3

说明:该程序中使用了最简单的 if 语句的格式。仅有 if 短语和一个 if 体,该 if 体用一个复合语句,它由3条赋值表达式语句组成,其功能是实现变量 a 和 b 之间的内容的交换。交换方法是通过第3个变量 t,先将 a 的值赋给 t,再将 b 的值赋给 a,最后将 t 的值赋给 b。其关系如下所示:



通过上述方法使得变量 a 和 b 的值进行了交换。这是一种有用的算法,以后还会用到。

如果要求将两个整数按由小到大的顺序输出时,只需将程序中

```
if(a<b)
```

改写为

```
if(a>b)
```

其余不变。这时上例中输出结果如下:

3, 5

[例4.2] 假设有一个物品其价格规定如下: 低于5公斤(含5公斤),每公斤 a 元,大于5公斤不足10公斤,每公斤 b 元,10公斤以上(含10公斤),每公斤 c 元。编程计算该物品为 m 公斤的价格。

```

main()
{
    float x,p,a,b,c;
    printf("Input weight x: ");
    scanf("%f",&x);
    printf("Input price a,b,c: ");
    scanf("%f%f%f",&a,&b,&c);
    if (x<=5)
        p=a*x;
    else if(x>5&& x<10)

```

```

        p=b * x;
    else
        p=c * x;
    printf("%.2f\n",p);
}

```

程序中变量 x 是该物品的重量, p 是该物品的价格。该程序中使用了一个 if 语句, 该语句含有一个 else if 短语和一个 else 短语。这是一个简单的多路分支程序。如果分支再多时, 可以再多用几个 else if 短语。

[例4.3] 比较两个整型数的大小。

```

main()
{
    int a,b;
    printf("Input a,b: ");
    scanf("%d%d",&a,&b);
    if (a!=b)
        if (a>b)
            printf("a>b\n");
        else
            printf("a<b\n");
    else
        printf("a=b\n");
}

```

执行该程序输入两个 int 型数:

```

Input a, b: -3 5
a<b

```

说明: 本例程序中, 用了一个 if-else 语句, 该语句的 if 体又是一个 if-else 语句, 这是 if 语句嵌套。当从键盘输入两个 int 型数以后, 程序中 a 和 b 获取了值。如, a 为 -3 , b 为 5 。显然, 这时 $a \neq b$ 为非零, 接着执行 if 体, 这又是一个 if-else 语句, 先判断 $a > b$ 为 0, 则执行 `printf("a < b\n");` 语句, 屏幕上显示 $a < b$ 。退出内重的 if 语句, 再退出外重 if 语句, 最后结束整个程序。

此例中使用的 if 语句还有其他形式。例如,

```

if (a==b)
    printf("a=b\n");
else
    if (a>b)
        printf("a>b\n");
    else
        printf("a<b\n");

```

还可以写成如下形式,

```

if (a==b)
    printf("a=b\n");
else if (a>b)
    printf("a>b\n");

```

```

else
    printf("a<b\n");

```

还可以写成如下形式:

```

if (a>=b)
    if (a==b)
        printf("a=b\n");
    else
        printf("a>b\n");
else
    printf("a<b\n");

```

还有很多其他的等价写法,这里不再一一列举。

在 if 语句嵌套时,如果 if 体或 else if 体或 else 体是一个 if 语句时,则应该被完全包含在体内。一般形式如下:

```

if (<条件1>)
    if (<条件2>)
        <语句1>
    else
        <语句2>
else
    if (<条件3>)
        <语句3>
    else
        <语句4>

```

这是一个 if-else 语句,该语句中 if 体和 else 体又是一个 if-else 语句,应按上述形式书写。

在使用 if 语句时应该特别注意的是 if 和 else 的配对关系。其原则如下:

一个 if 语句中最多只能有一个 else 短语(可以没有);以最内层开始(如有嵌套),else 总是与它前面最近的未曾配对的 if 配对。

[例4.4] 分析下列程序输出结果。

```

main()
{
    int a,b;
    a=b=2;
    if (a!=1)
        if (b==1)
            printf("%d\n",a+b);
        else
            printf("%d\n",a-b);
    printf("%d\n",a+b);
}

```

该程序从表面分析看好像是一个 if-else 语句,而 if 体是一个 if 语句,这样分析输出结果是4。因为首先 $a \neq 1$ 为非零,又 $b == 1$ 为零,则退出整个 if 语句,执行 $\text{printf}("%d\n", a+b);$

输出结果为4。而实际上,按照 else 的配对原则,该例中 else 应该与第二个 if 配对,if 语句应书写如下:

```
if (a!=1)
    if (b==1)
        printf("%d\n",a+=b);
    else
        printf("%d\n",a-=b);
```

这实际上是一个简单的 if 语句,该 if 语句的 if 体是一个 if-else 语句。按这种情况,分析输出结果为:

由此可见,书写的对齐格式往往会产生误导,一定要注意书写格式正确性,同时应该牢记 else 的配对规则,这样可以避免“误导”。

如果该程序中,确要 else 短语与第一个 if 配对,那么该段程序作如下修改:

```
if (a!=1)
{
    if (b==1)
        printf("%d\n",a+=b);
}
else
    printf("%d\n",a-=b);
```

这里通过加上一对花括号来改变其配对关系。

4.3.2 开关语句

开关语句也是一种根据条件来判断的多路分支选择语句。开关语句又称为 switch 语句。

1. 开关语句的格式和功能

开关语句的格式如下:

```
switch (<整型表达式>)
{
    case <整型表达式1>: <程序段1>
    case <整型表达式2>: <程序段2>
    case <整型表达式3>: <程序段3>
        :
    case <整型表达式 N>: <程序段 N>
    default: <程序段 N+1>
}
```

其中,switch,case 和 default 是关键字。<整型表达式>要求其表达式的值为 int 型数,否则转换成 int 型数。<整型表达式1>至<整型表达式 N>要求表达式用 int 型常量组成,不得含有变量,一般用数字或字符组成。<程序段1>至<程序段 N+1>分别是由0条、1条或多条语句组成的语句序列。一般情况下,按其需要非空程序段的最后一条语句用 break;,表示退出该开关

语句,执行其后面语句。如果该程序段中没有 break; 语句,则表示继续执行下面的程序段,直到遇到 break; 语句或所有程序段都执行完再退出该开关语句。

开关语句执行过程如下:

先计算 switch 关键字后面的〈整型表达式〉的值,并转换成 int 型数。然后用该值依次顺序地与花括号内的 case 关键字后面的整型表达式的值进行比较。先与〈整型表达式1〉的值进行比较,如果相等,则执行〈程序段1〉,遇到 break; 语句退出该开关语句,否则继续执行〈程序段2〉,直到遇到 break 语句退出该开关语句为止;如果不等,将〈整型表达式〉的值再与〈整型表达式2〉的值进行比较,如果相等,则执行〈程序段2〉,其方法同于执行〈程序段1〉;如果不等,再将〈整型表达式〉的值与〈整型表达式3〉的值进行比较,依次类推。总之,将 switch 语句开头的整型表达式的值依次与花括号内 case 后边的整型表达式的值进行比较,哪个整型表达式的值与其相等,便开始执行该整型表达式之后对应的程序段,如果都不相等,则执行关键字 default 后面的程序段,如果没有 default,则什么都不做,退出该 switch 语句。当一旦开始执行某个程序段时,当遇到 break 语句后,则立即退出该 switch 语句,如果在整个程序段中没有 break 语句,则继续执行该程序段下面的程序段,同样是遇到 break 语句退出该 switch 语句,否则再继续执行下面的程序段,直到执行完最后一个程序段遇到右花括号为止,这时退出该 switch 语句,这便是 break 语句在开关语句中的作用。

2. 开关语句的应用和说明

开关语句是一个很好的多路分支选择语句,常用它来实现较为复杂的多路分支程序。下面将通过一些例子讲述开关语句的具体应用。

[例4.5] 编写一个程序,实现两个浮点数的四则运算。程序中,用 d1和 d2表示两个浮点数,用 op 表示四则运算的运算符: +, -, *, /。

程序内容如下:

```
main()
{
    float d1,d2;
    char op;
    printf("Input d1,op,d2 :");
    scanf("%f,%c,%f",&d1,&op,&d2);
    switch(op)
    {
        case '+':printf("%.2f%c%.2f= %.2f\n",d1,op,d2,d1+d2);
                break;
        case '-':printf("%.2f%c%.2f= %.2f\n",d1,op,d2,d1-d2);
                break;
        case '*':printf("%.2f%c%.2f= %.2f\n",d1,op,d2,d1*d2);
                break;
        case '/':printf("%.2f%c%.2f= %.2f\n",d1,op,d2,d1/d2);
                break;
        default: printf("op is error!\n");
    }
}
```

执行该程序出现下列提示信息:

Input: d1, op, d2: 3.5, *, 4 ✓

按要求键入数据后,执行程序输出结果为:

3.50 * 4.00 = 14.00

说明: 该例中使用了 switch 语句,共有5个分支,除了+, -, *, /外,还有一个不可识别的表达式。每个分支对应着一个程序段,除最后一个程序段外,其余4个程序段中都有 break 语句,这意味着执行完某个程序段后都退出 switch 语句,最后一个程序段没有 break 语句,实际上执行完该程序段后遇上了右花括号,也有退出 switch 语句的作用,因此最后一个程序段可以省略 break 语句。

[例4.6] 编写一个程序用来统计从键盘上输入的字符流中,每种数字字符、空白字符和其他字符出现的个数。

程序中将每种数字字符统计的个数存放在一个 int 型数组 ndigit 中,该数组共10个元素,用 ndigit[0], ndigit[1], ... 分别存放数字字符0, 1, ... 的个数。空白字符(包含空格符、换行符和水平制表符)个数存放在变量 nwhite 中,其他字符的个数存放在变量 nother 中。

程序内容如下:

```
#include <stdio.h>
main()
{
    int i, c, ndigit[10], nwhite, nother;
    nwhite = nother = 0;
    for(i = 0; i < 10; i++)
        ndigit[i] = 0;
    while((c = getchar()) != EOF)
        switch(c)
        {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9': ++ndigit[c - '0'];
                    break;
            case ' ':
            case '\t':
            case '\n': ++nwhite;
                    break;
            default: ++nother;
        }
    printf("digiter = ");
    for(i = 0; i < 10; i++)
        printf(" %d ", ndigit[i]);
    printf("\nwhitespace = %d, other = %d\n", nwhite, nother);
}
```

```
}
```

执行该程序从键盘上输入如下字符流：

ab35795\trmn45379_ pq98765 <Ctrl>z ↵

其中,<Ctrl>z↵是用来结束输入的字符流。

屏幕显示如下信息：

digiter=0002141313

whitespace=2, other=6

说明：该程序中用到了后面要讲述的循环语句：for 循环和 while 循环，这里不详细讲解，可参照下面将要讲述的关于循环语句的内容。该程序中使用了一个 switch 语句作为 while 循环的循环体，该 switch 语句中共有14个分支，仅有3个非空程序段。这里，有些分支对应着空程序段，例如，对于数字字符的10个分支只有一个非空程序段，这意味着凡是输入数字字符都执行 case '9'：后面对应的程序段，这样将为书写带来方便。同样，凡是输入空白符时都执行 case '\n'：后面对应的程序段。空程序段的使用是该例的特点。该例中，对数字字符和空白符的统计使用不同的 case 分支来实现，而对其他字符的统计使用 default 最为合适。

该例中的 switch 语句也可以用 if 语句来替代，替代这里的 switch 语句的 if 语句格式如下：

```
if (c>='0' && c<='9')
    ++ndigit[c-'0'];
else if (c=='_')
    ++nwhite;
else if (c=='\t')
    ++nwhite;
else if (c=='\n')
    ++nwhite;
else
    ++nother;
```

请读者试一下，这样替代后看其结果是否相同。

switch 语句中要求 case 短语后面是一个整常型表达式，一般只能用整数或字符，而不允许使用关系表达式或逻辑表达式。因此，在有些情况下，需要将某种较复杂的关系转换为简单的数字，以便使用 switch 语句来编程，下面便是一个这方面的例子。

[例4.7] 有一商场为了促销对某商品实行打折销售。具体办法如下：（假设购买商品个数为 x ）

$x < 5$	没折扣
$5 \leq x < 10$	1%折扣
$10 \leq x < 20$	2%折扣
$20 \leq x < 30$	4%折扣
$30 \leq x$	6%折扣

假定该商品单价为 m 元，编程计算某顾客购买 x 个该商品应付多少钱。计算公式如下：

$p = m * x * (1 - d)$

其中， p 为应付钱数（元）， d 是所打的折扣数。

程序内容如下：

```
main()
{
    int a,x;
    float p,m,d;
    printf("Input m,x: ");
    scanf("%f%d",&m,&x);
    if (x>30)
        a=6;
    else
        a=x/5;
    switch(a)
    {
        case 0:d=0.0;
            break;
        case 1:d=0.01;
            break;
        case 2:
        case 3:d=0.02;
            break;
        case 4:
        case 5:d=0.04;
            break;
        case 6:d=0.06;
    }
    p=m * x * (1-d);
    printf("%.2f\n",p);
}
```

执行该程序,显示如下提示信息:

Input m, x: 25.5 28 ✓

输出结果如下:

685.44

说明:分析题意,可看出折扣的变化是有规律的,其“变化点”是5的倍数,利用这一特点将购买的商品数目除以5,则分出不同折扣的档次,于是可以使用 switch 语句了。

本例中所用的 switch 语句共7个分支,5个程序段,无 default 短语。

在开关语句中,default 短语可以有,也可以没有。如果有 default 短语,它可以放在开关语句的花括号内的任一位置,无论放在何处它的功能都是不变的。default 后面的程序段只有在所有的整型表达式的值都与前面的整型表达式的值不相等时才被执行。这一功能与 default 的位置无关。

[例4.8] 分析下列程序的输出结果。

```
#include <stdio.h>
main()
{
    char c;
```

```

int x,y,z;
x=y=z=0;
while((c=getchar())!='\n')
    switch(c)
    {
        case 'a':
        case 'b':
        case 'c': x++;
                break;
        default: z++;
                break;
        case 'A':
        case 'B':
        case 'C': y++;
    }
printf("x=%d,y=%d,z=%d\n",x,y,z);
}

```

当输入如下信息时，

aBcDeF ↵

输出结果如下：

x=2, y=1, z=3

说明：该程序的 switch 语句中有 default 短语，它没有放在开关语句的最后，而是放在 case 短语的中间，即前后各有3个 case 短语。读者可以验证一下，该程序中 switch 语句的形式与下述两种形式是等价的。

形式一：

```

switch(c)
{
    case 'a':
    case 'b':
    case 'c': x++;
            break;
    case 'A':
    case 'B':
    case 'C': y++;
            break;
    default: z++;
}

```

形式二：

```

switch(c)
{
    default: z++;
            break;
    case 'a':
    case 'b':

```

```

    case 'c': x++;
        break;
    case 'A':
    case 'B':
    case 'C': y++;
}

```

可见这几种不同形式的 switch 语句只是区别在 default 短语的位置上。另外,凡是最后的一个程序段都可以省去 break 语句。

4.4 循环语句

C 语言中提供了构成循环结构的循环语句。C 语言提供了三种循环语句: while 循环语句、do-while 循环语句和 for 循环语句。它们各自有其特点,并且有些可以相互转换。

4.4.1 while 循环语句

1. while 循环语句的格式和功能

while 循环语句的格式如下:

```

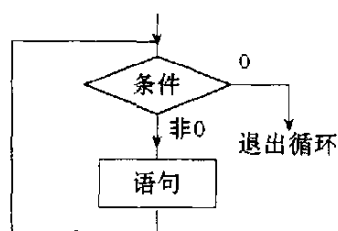
while (<条件>)
    <语句>

```

其中,while 是关键字,<条件>是通过一种表达式给出的是否执行循环体的判断条件,当表达式的值为非零时,执行循环体;否则不执行循环体,退出循环,执行该循环后面的语句。常用的作为条件的表达式是关系表达式或逻辑表达式,也可以用其他表达式或常量。当使用赋值表达式时,有些编译出现警告错。<语句>是循环体,它可以是一条语句,也可以是复合语句。如果在 while 循环头下面有花括号,则循环体将是由花括号括起的复合语句。如果在 while 循环头下面无花括号,则循环体将是一条语句,其余语句是循环语句后面的语句。

while 循环语句执行过程如下:

先计算由条件给出的表达式的值,如果该表达式的值为非零,则执行循环体的语句;否则退出循环,不执行循环体,而执行该循环语句后面的语句。while 语句的执行过程可由下图表示:



可见,该种循环可能由于条件不满足一次循环体都不执行。

2. while 循环语句的应用和说明

下面通过具体实例进一步说明 while 循环语句的应用。

[例4.9] 编程计算1~100自然数之和。

```

main()
{
    int i=1,sum=0;
    while(i<=100)
    {
        sum+=i;
        i++;
    }
    printf("%d\n",sum);
}

```

执行该程序输出结果如下：

5050

说明：该程序中，循环条件是一个关系表达式 $i \leq 100$ ，当该表达式值为1时，执行循环体，当表达式值为0时，退出循环，执行循环语句后面的 `printf()` 语句。该循环的循环体是一个复合语句，因为它是由两条语句组成的。该循环体可以进一步简化为由一条语句组成，即可写成如下形式：

```

while (i<=100)
    sum+=i++;

```

请读者验证一下，这种形式是正确的。

4.4.2 do-while 循环语句

1. do-while 循环语句的格式和功能

do while 循环语句的格式如下：

```

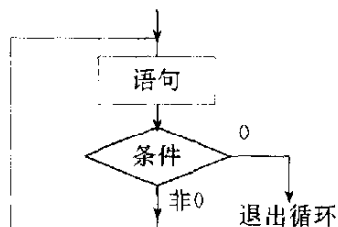
do (语句)
while (<条件>);

```

其中，do 和 while 是关键字。<语句>是循环体，它可以是一条语句，也可以是复合语句。<条件>将由任一表达式给出，常用的是关系表达式或逻辑表达式，也可以是其他表达式和常量。如果是赋值表达式，有些编译会发生警告错。

do-while 语句的执行过程如下：

先执行一次循环体的语句，再计算条件所指定的表达式的值，如果表达式值为非零，则再执行循环体，否则退出循环，执行该循环语句后面的语句。该种循环不管条件如何，总是要执行一次循环体的，这便是 do-while 循环语句的特点。该语句的执行过程由下图所示：



do-while 循环语句可以用前面讲过的 while 循环语句表示,具体格式如下:

```
<语句>
while (<条件>)
    <语句>
```

2. do-while 语句的应用和说明

下面举例说明 do-while 语句的应用。

[例4.10] 编程计算1~100自然数之和。

```
main()
{
    int i=1,sum=0;
    do {
        sum+=i++;
    }while(i<=100);
    printf("%d\n",sum);
}
```

执行该程序输出结果与例4.9程序的输出结果相同。

说明:使用 do-while 语句时,为了提高可读性,在书写时总是习惯于将循环体用一对花括号括起来,while 关键字写在右花括号的后面,就是单条语句作循环体也加上花括号,本例就是如此。另外,需要注意的是在 while 关键字后面的(<条件>)后面要加一个分号,该分号不可省去。

该例中 do-while 循环语句可写成如下的 while 语句的格式:

```
sum+=i++;
while (i<=100)
    sum+=i++;
```

请读者注意不是所有的 do-while 语句都可写成上述的 while 语句形式。

while 循环语句与 do-while 循环语句是可以嵌套的,即在 while 循环体中可以出现 do-while 语句,反之,也可以。下面是一个循环语句嵌套的例子。

[例4.11] 分析下列程序的输出结果。

```
main()
{
    int i=5;
    while(i-->0)
    {
        do {
            printf("%4d",i--);
        }while(i>2);
        i++;
    }
    printf("\n");
}
```

执行该程序输出结果如下:

4 3 2 1 0

说明：本例程序中，while 循环语句的循环体中有一个 do-while 语句，这便是循环语句的嵌套。执行该程序，作 while 循环时， $i-->0$ 为 1，则执行循环体，再执行 do-while 语句时，先作循环体，输出 $i--$ 的值为 4，同时 i 变成 3，判断循环条件时， $i>2$ 为 1，又执行一次 do-while 循环体，输出 $i--$ 的值为 3，同时 i 变成 2，再判断循环条件时， $i>2$ 值为 0，退出 do-while 循环，执行 $i++$ 语句，这时 i 的值为 3，再判断 while 的循环条件时， $i-->0$ 值为 1，这时 i 变成 2，再作 do-while 循环的循环体，输出 $i--$ 的值为 2，而 i 变成 1，判断 do-while 循环条件时， $i>2$ 值为 0，接着执行 $i++$ 后， i 变成 2。再判断 while 循环条件时， $i-->0$ 值为 1，这时 i 变成 1，再作 do-while 循环的循环体，输出 $i--$ 的值为 1，而 i 变成 0，判断 do-while 循环条件时， $i>2$ 值为 0，接着再执行 $i++$ 后， i 变成 1。再判断 while 循环条件时， $i-->0$ 为 1， i 变成 0，再作 do-while 循环的循环体，输出 $i--$ 的值为 0，而 i 变成 -1，判断 do-while 循环条件时， $i>2$ 值为 0，接着执行 $i++$ 后， i 变成 0。再判断 while 循环条件时， $i-->0$ 的值为 0，则退出 while 循环语句，结束该程序。所以，多次循环输出结果为 4 3 2 1 0。这里，while 语句共循环 4 次，do-while 循环语句除了第一次 while 循环时执行 2 次循环体（输出 4 和 3）外，以后 3 次 while 循环时，它每次仅执行一次循环体，分别输出 2 1 和 0。

4.3.3 for 循环语句

1. for 循环语句的格式和功能

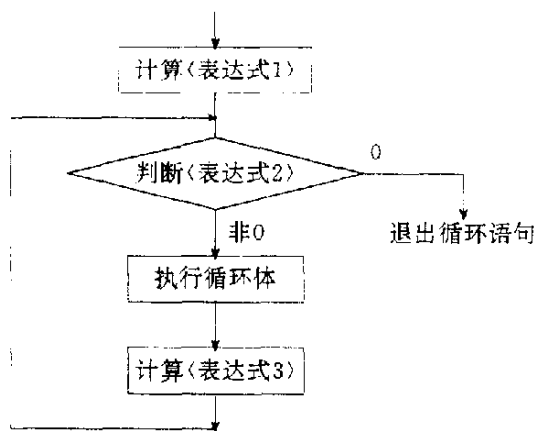
for 循环语句的格式如下：

```
for (<表达式1>; <表达式2>; <表达式3>)
    <语句>
```

其中，for 是关键字。<表达式1>、<表达式2>和<表达式3>是任意表达式，也可以是逗号表达式；<语句>为循环体，它可以是一条语句，也可以是复合语句，还可以是空语句。

for 循环语句的执行过程如下：

先计算<表达式1>的值，再计算<表达式2>的值。接着进行判断，如果<表达式2>的值为零，则退出该循环语句，执行该循环语句后面的语句。如果<表达式2>的值为非零，则执行循环体的语句，再计算<表达式3>的值。然后再计算<表达式2>的值，再进行判断是退出循环还是继续执行循环体，重复上述操作，直到退出循环语句为止。for 循环语句执行过程如下图所示：



for 循环语句也可以写成 while 循环语句的形式。下面便是由 while 循环语句写成的 for 循环语句形式：

```
d1;  
while(d2)  
{  
    <语句>  
    d3;  
}
```

因此,任一的 for 循环语句格式都可用 while 循环语句来替代。反之,也可以。

2. for 循环语句的应用和说明

for 循环语句具有以下特点：

(1) 已知循环次数的循环使用该语句比较方便,对于事先难以确定循环次数的循环用此语句有时不方便。

(2) 该种循环语句格式比较灵活,<表达式1>、<表达式2>和<表达式3>可以移出 for 循环语句循环头后面的圆括号,而放到其他合适位置。表达式移出后,其分号仍然保留,不得省去。在一般情况下,<表达式1>给出循环变量的初始值,它只计算一次,有时可放在 for 循环语句之前;<表达式2>给出判断是否循环的条件,该表达式为非零时,执行循环体,否则退出循环体;<表达式3>给出循环变量的改变量,即增量,每次执行完循环体后,都要计算该表达式,改变循环变量的值。另外,这三个表达式都可以是逗号表达式。

下面通过具体实例来说明 for 循环语句的特点。

[例4.12] 编程计算1~10自然数之和。下面给出5种不同格式的 for 循环语句,各程序执行结果都是相同的。

程序一：

```
main()  
{  
    int i,sum=0;  
    for(i=1;i<=10;i++)  
        sum+=i;  
    printf("%d\n",sum);  
}
```

程序二：

```
main()  
{  
    int i=1,sum=0;  
    for(;i<=10;i++)  
        sum+=i;  
    printf("%d\n",sum);  
}
```

程序三：

```
main()
```

```

{
    int i=1,sum=0;
    for(;i<=10;)
        sum+=i++;
    printf("%d\n",sum);
}

```

程序四：

```

main()
{
    int i=1,sum=0;
    for(;;)
    {
        sum+=i++;
        if(i>10)
            break;
    }
    printf("%d\n",sum);
}

```

程序五：

```

main()
{
    int i,sum;
    for (i=1,sum=0;i<=10;sum+=i,i++)
        ;
    printf("%d\n",sum);
}

```

说明：从该例中可以看出 for 循环的各种不同的格式，使用中可以灵活选择。

(1) for 语句一般形式中的〈表达式1〉可以省略，此时〈表达式1〉应出现在循环语句之前，正如本例程序二所示。

(2) for 语句一般形式中的〈表达式2〉如果省略，即不判断其循环条件，则循环将无终止地进行下去。要结束循环，需要在循环体内设置退出循环的语句 break，正如本例程序四所示。

(3) for 循环语句一般形式中的〈表达式3〉可以省略，即将循环变量增量的操作放在循环体内进行，本例中程序三便属于此类情况。

(4) 〈表达式1〉不仅可以设置循环变量的初值，同时还可以通过逗号表达式设置一些其他值。类似地，〈表达式2〉和〈表达式3〉都可以这样，正如本例中程序五所示。

for 循环可以与其他循环嵌套，它自身也可以嵌套，下面举例说明。

[例4.13] 编程求出下列式子中 M 和 N 的值。

$$\begin{array}{r}
 M \quad N \\
 \times \quad N \quad M \\
 \hline
 6 \quad 7 \quad 8 \quad 6
 \end{array}$$

程序内容如下：


```

main()
{
    int m,n,k;
    for(m=1;m<10;m++)
        for(n=1;n<10;n++)
        {
            k=(10*m+n)*(10*n+m);
            if(k==6786)
                printf("M=%d,N=%d\n",m,n);
        }
}

```

执行该程序输出结果如下：

M=7, N=8

M=8, N=7

说明：该程序中使用了 for 循环语句的嵌套，即用 for 循环语句作为 for 循环的循环体。这称为双重 for 循环。前边一个称为外重循环，后面一个称为内重循环。

〔例4.14〕 编程打印出下列图案。

```

      *
     * * *
    * * * * *
   * * * * * *
  * * * * * * *
 * * * * * * *
* * * * * * *
 * * * * * *
  * * * * *
   * * * *
    * * *
     *

```

程序内容如下：

```

main()
{
    int i,j;
    for(i=1;printf("\n"),i<=5;i++)
        for(j=1;j<=i+7;j++)
            if(j<=7-i)
                printf(" ");
            else
                printf(" * ");
    for(i=4;i>=1;i--)
    {
        for(j=1;j<=i+6;j++)
            if(j<=7-i)
                printf(" ");
            else
                printf(" * ");
        printf("\n");
    }
}

```

说明：该程序中出现了两次 for 的双重循环，第一次 for 的双重循环打印出5行图案的上

部分,第二次 for 的双重循环打印出4行图案的下部分。另外,在最前面的一个 for 循环语句中,〈表达式2〉是一个逗号表达式,其中 printf("\n"); 语句用来实现每行换行的,在打印图案下半部分时,printf("\n"); 语句放在外重循环的循环体内来实现每行换行的。

[例4.15] 编程求出50至100之间的全部素数。

所谓素数是只能被1和本身整除的数。求解方法很多,这里仅选用一种算法简单、效率较低的方法。

程序内容如下:

```
main()
{
    int i,j,n=0;
    for(i=51;i<=100;i++)
    {
        j=2;
        while(i%j!=0)
            j++;
        if(j==i)
        {
            if(n%6==0)
                printf("\n");
            n++;
            printf("%6d",i);
        }
    }
    printf("\n");
}
```

执行该程序输出如下结果:

```
53    59    61    67    71    73
79    83    89    97
```

说明:该程序中 for 循环体内出现了 while 循环,这也是循环的嵌套。

求素数的方法很多,下面再给出一种比该程序中所使用的方法效率要高些的求素数方法,将上述程序重写如下:

```
#include <math.h>
main()
{
    int i,j,k,n=0;
    for(i=51;i<=100;i+=2)
    {
        k=sqrt((double)i);
        for(j=2;j<=k;j++)
            if(i%j==0)
                break;
        if(j>=k+1)
        {
            if(n%6==0)
```

```

        printf("\n");
        n++;
        printf("%6d", i);
    }
}
printf("\n");
}

```

读者可以与前面程序比较一下,后一种方法程序中循环次数将会减少很多,于是将提高运算速度,但是这种求素数算法难懂一些,这里不去证明这一算法的正确性,读者可以选用。

C 语言所提供的三种循环语句(while 循环、do-while 循环和 for 循环),它们自身可以嵌套,它们相互之间也可以嵌套。嵌套时应该注意的是要在一个循环体内包含另一个完整的循环结构,这就是说,无论哪种嵌套关系都必须将一个完整的循环结构全部放在某个循环体内。下面举几个合法嵌套的格式:

(1) do-while 循环的自身嵌套。for 循环和 while 循环也类似。

```

do {
    :
    do {
        :
    } while (...);
    :
} while (...);

```

(2) for 循环内嵌套 while 循环。

```

for (;;)
{
    :
    while(...)
    {
        :
    }
    :
}

```

(3) while 循环内嵌套 do-while 循环。

```

while (...)
{
    :
    do {
        :
    } while (...);
    :
}

```

(4) for 循环内嵌套 while 循环和 do-while 循环。

```

for (;;)

```

```

{
    :
    while (...)
    {
        :
    }
    do {
        :
    } while (...);
    :
}

```

其他的各种循环嵌套将不一一列举。

4.5 转向语句

C 语言提供4种转向语句,它们是 goto 语句、break 语句、continue 语句和 return 语句。下面分别讲述这些语句的格式、功能和用法。

4.5.1 goto 语句

goto 语句是无条件转向语句,它的格式如下所示:

```
goto <语句标号>;
```

其中, goto 是关键字,<语句标号>是一种标识符,按标识符的规则来写出语句标号。语句标号是用来标识一条语句的,这种标识专门给 goto 转向语句使用的,即指明 goto 语句所要转向的语句。语句标号出现在语句的前面,用冒号(:)与语句分隔。其格式如下所示:

```
<语句标号>: <语句>
```

一条语句可以有一个或多个语句标号,多数语句不带语句标号,只有 goto 语句需要转向到的语句才加语句标号。由于 C 语言中对 goto 语句采取限制使用的方法,限制 goto 语句转向只能在本函数体内。因此语句标号要求在一个函数体内是唯一的,不同函数体可以相同,所以,语句标号的作用范围也被限制在本函数体内。

在 C 语言程序中尽量要少用 goto 语句,最好不用 goto 语句,因为它会破坏结构化,影响可读性。goto 语句最常见的用法一是用来与 if 语句构成循环结构,二是用来以多重循环最内重一次退到最外边。在使用 goto 语句时,要注意在转向时越过循环语句的循环头和分程序的说明语句部分时,可能会出现错误,请要小心慎重。

下面通过程序实例说明 goto 语句的应用。

[例4.16] 使用 goto 语句与 if 语句构成循环计算1至100自然数之和。

程序内容如下:

```

main()
{
    int i=1,sum=0;
loop: if(i<=100)
    {

```

```

        sum+=i++;
        goto loop;
    }
    printf("%d\n",sum);
}

```

执行该程序输出结果如下：

5 0 5 0

[例4.17] 编写一个程序,从已知的二维数组中查找第一次出现负的数组元素。

```

main()
{
    int i,j,num[2][3];
    printf("Input 6 digiters: ");
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            scanf("%d",&num[i][j]);
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            if(num[i][j]<0)
                goto found;
    printf("not find!\n");
    goto end;
found: printf("num[%d][%d]=%d\n",i,j,num[i][j]);
end:    ;
}

```

说明：该程序中,两次使用双重 for 循环,第一次是通过键盘输入数据给二维数组 num[2][3]赋值,第二次使用双重 for 循环是从数组中按顺序一个一个查找其值为负的元素。如果找到后,则将该数组元素的值输出;如果找不到则输出信息“not find!”。

该程序中,使用了二次 goto 语句,于是在两个语句前出现了语句标号: found 和 end。其中,end 所标识的是一个空语句。

4.5.2 break 语句

break 语句格式如下:

```
break;
```

其中,break 是关键字。该语句的功能有两个:

(1) 在 switch 语句的程序段中,遇到 break 语句,则退出 switch 语句,执行开关语句后面的语句。

(2) 在循环体中,遇到 break 语句,则退出该重循环。这里所说的该重循环是指在多重循环中,在最内重循环中,遇到 break 语句,只退该语句所在那重循环;到它的外重循环,还需再用 break 语句,才能退出另一重循环。如果有三重循环,需要在不同重循环中分别使用3个 break 语句才能退出三重循环。

下面举例说明 break 语句在循环体中的应用。

[例4.18] 分析下列程序的输出结果。

```

main()
{
    int i=1;
    do {
        i++;
        printf("%d\n", ++i);
        if (i==7)
            break;
    }while(i==3);
    printf("ok!\n");
}

```

执行该程序输出结果如下：

```

3
5
ok!

```

说明：该程序的 do-while 循环中，在 if 语句里使用了 break 语句，即当 $i==7$ 为非 0 时，执行 break 语句，退出 do-while 循环。本例中 do-while 循环共执行二次循环体，分别输出 3 和 5。

4.5.3 continue 语句

continue 语句格式如下：

```
continue;
```

其中，continue 是关键字。该语句的功能只是用在循环体中，执行该语句则结束本次循环，再去判断是否继续下次循环。该语句也是用于循环体中 if 语句内，即满足某种条件才结束本次循环。

下面举例说明 continue 语句的应用。

[例4.19] 编程输出 1 至 100 之间能被 17 整除的自然数。

程序内容如下：

```

main()
{
    int i;
    for(i=1; i<=100; i++)
    {
        if (i%17!=0)
            continue;
        printf("%4d", i);
    }
    printf("\n");
}

```

执行该程序输出如下结果：

```
17   34   51   68   85
```

说明：该程序在 for 循环的循环体内的 if 语句中使用了 continue 语句，当执行到该语句时，结束本次循环，即该语句后面的循环体中的语句不再执行，而直接执行 $i++$ 和 $i<=100$ 。

判断是否还要继续执行循环体。

就该例而言,为了实现题中的要求,for 循环的循环体也可以不用 continue 语句,而写成如下形式。

```
for (i=1; i<=100; i++)
    if (i%17==0)
        printf("%4d", i);
```

这样书写更简单些。

[例4.20] 分析下列程序的输出结果。

```
main()
{
    int i, j;
    i=j=1;
    for (; i<=50; i++)
    {
        if (j>15)
            break;
        if(j%3==1)
        {
            j+=4;
            continue;
        }
        j+=3;
    }
    printf("%d\n", i);
}
```

执行该程序输出结果如下:

6

说明: 该程序的 for 循环中, 分别在 if 语句里使用了 break 和 continue 语句。开始时, i 和 j 都为1, 每执行一次 for 循环体, i 则加1。第一次执行 for 循环体时, 由于 j%3==1 值为1, 则 j 被加4, 以后再执行 for 循环体时, j 被加3, 待到 j>15 时, 退出 for 循环, 这时共执行6次循环体, 因此, 最后输出 i 为6。

4.5.4 return 语句

return 语句又称为返回语句。该语句的格式如下:

```
return;           和
return(<表达式>);
```

其中, return 是关键字。该语句有上述两种格式: 不带返回值格式和带返回值格式。该语句用在被调用函数中, 在被调用函数中执行到该语句时, 将结束对被调用函数的执行, 并把控制权返回给调用函数, 继续执行调用函数后边的语句。在带有返回值的情况下, 将 return 语句所带的表达式的值返回给调用函数, 作为调用函数的值。

在被调用函数中, 可以用 return 语句, 也可以不用 return 语句。如果要求被调用函数有返回值, 则一定要用 return 语句, 采用 return(<表达式>) 格式。如果被调用函数不需要返回值, 并

且当被调用函数的所有语句执行完后进行返回,则被调用函数可以不用 `return` 语句,函数体的右花括号具有返回语句的作用。函数的返回值可以用来实现函数之间的信息传递。在 C 语言中,实现函数之间信息传递的方式有三种,返回值是其中的一种,但是这种传递信息方式的局限性是每次只能传递一个信息。

有关返回语句的应用和返回值的一些规定将在函数一章中还会讲到,这里不再举例。

练 习 题

1. C 语言提供了哪些种类的语句?使用这些语句能否构成结构化程序设计的三种基本结构?
2. 表达式和表达式语句二者之间有何区别?
3. 什么是空语句?它有何作用?
4. 什么是复合语句?它与分程序有何区别?
5. 条件语句(`if` 语句)格式是如何规定的?其中,条件是如何给出的?`else` 短语在使用中有何规定?
6. 开关语句(`switch` 语句)格式是如何规定的?其中,`default` 短语在使用中有何规定? `break` 语句起什么作用?
7. `while` 循环语句、`do-while` 循环语句和 `for` 循环语句各自的格式如何?相互之间有何区别?在具体编程时如何选择?
8. 什么叫循环的嵌套?C 语言中所提供的三种循环都可以相互嵌套使用吗?循环嵌套应注意些什么问题?
9. `break` 语句和 `continue` 语句在循环语句的循环体中各起什么作用?
10. `goto` 语句的使用格式如何?语句标号是如何规定的?
11. `return` 语句的格式和功能如何?
12. 总结 C 语言中所提供的各种语句的格式、功能和使用方法。

作 业 题

1. 判断下列描述的正确性,对者划√,错者划×。
 - (1) 分程序是一种复合语句,而不是所有的分程序都是复合语句。
 - (2) 使用 `if-else` 语句可以实现多路分支,因为对 `else if` 短语的个数不限,并且 `if` 体、`else if` 体和 `else` 体内还可以嵌套 `if` 语句。
 - (3) 在多个 `if-else` 语句嵌套使用时,从最内层开始,`else` 总是与它上面最近的一个 `if` 配对,并且一个 `if` 仅能有一个 `else`。
 - (4) 在 `switch` 语句中,多个 `case` 短语不可共用一个程序段。
 - (5) 在 `switch` 语句中,每个 `case` 后面的整常量表达式的值可以相同。
 - (6) `switch` 语句,必须有且仅有一个 `default` 短语,否则该语句会出错。
 - (7) `for` 循环语句中,当 `for` 短语后面三个表达式中,中间的表达式(即<表达式2>)省略,则意味着该表达式的值为0。
 - (8) `continue` 语句在循环体中可用来结束本次循环,而不是退出循环。
 - (9) 语句标号在同一个文件中是唯一的,不同文件可以相同。
 - (10) 使用 `break` 语句是退出 `switch` 语句的唯一方法。

2. 选择填空

(1) 在 C 语言的 while 循环语句中, 用作条件的表达式为()。

- A. 任意表达式
- B. 算术表达式
- C. 赋值表达式
- D. 逗号表达式

(2) 下列()是语句。

- A. printf("OK!\n")
- B. x+y
- C. a=5
- D. ;

(3) 已知: int i,x; 下列 for 循环的循环次数是()。

```
for (i=0, x=0; !x&& i<=5; i++) x++;
```

- A. 5次
- B. 6次
- C. 1次
- D. 无限

(4) 已知: int i=5; 下述 while 循环执行()次。

```
while(i=0) i--;
```

- A. 0
- B. 1
- C. 5
- D. 无限

(5) 已知: int i=5; 下述 do-while 循环执行()次。

```
do { printf ("%d\n", i--);  
    i--;  
} while (i!=0);
```

- A. 0
- B. 1
- C. 5
- D. 无限

(6) 已知 int i; 下列 for 循环执行循环体()次。

```
for (i=0, j=10; i=j=10; i++, j--)  
    printf("OK!\n");
```

- A. 0
- B. 1
- C. 10
- D. 无限

(7) 用()语句退出循环体是错误的。

- A. return
- B. goto
- C. break
- D. continue

(8) 在开关语句中, case 短语后边的表达式使用()是错误的。

- A. 字符常量
- B. 数字常量
- C. 符号常量
- D. 任意表达式

(9) 下列关于循环体的描述中,()是错的。

- A. 循环体中不仅可出现循环语句还可出现选择语句。
- B. 循环体中可以出现 break 语句和 continue 语句。
- C. 循环体中不能出现 goto 语句。
- D. 循环体中可以出现复合语句也可出现分程序。

(10) 下列关于循环条件的描述中,()是错的。

- A. 常量和变量都可以作为循环条件。
- B. 逗号表达式可作为循环条件。
- C. 关系表达式和关系表达式语句都可作为循环条件。
- D. while 循环条件不能省略。

3. 分析下列程序的输出结果。

(1)

```
main()  
{  
    int i,j;  
    for(i=j=0; i<=5, j<=3; i++, j++)
```

```

        printf("ok!");
        printf("ok\n");
    }

```

(2)

```

main()
{
    int x,y;
    x=5;
    y=x++-3;
    if (x++>y+10)
        printf("%d",x);
    else
        printf("%d\n",x++);
}

```

(3)

```

main()
{
    int i=-1,j=3,c;
    do {
        c=(++i<0)&&!(--j==0);
        i++;
    }while(c);
    printf("%d,%d\n",i,j);
}

```

(4)

```

main()
{
    int s[5],i,j,k;
    for(i=0;i<5;i++)
        s[i]=0;
    for(k=3,i=0;i<k;i++)
        for(j=0;j<k;j++)
            s[j]=s[i]+j;
    printf("%d\n",s[2]);
}

```

(5)

```

main()
{
    int i=3,x=3,y=4;
    do {
        i*=10+x++-y--;
    }while(i<20);
    printf("%d\n",i);
}

```

(6)

```

main()

```

```

{
    int i,j;
    for(i=0;i<5;i++)
    {
        switch(i)
        {
            case 4: printf("4");
            case 3: printf("3");
            case 2: printf("2");
            case 1: printf("1");
            case 0: printf("0\n");
        }
        for(j=0;j<5;j++)
            printf(" * ");
        printf("\n");
    }
}

```

(7)

```

main()
{
    int c,i=0;
    static char s[]="abc";
    c=s[0];
    while(c)
    {
        switch(c)
        {
            case 'a':
            case 'b': printf("%d",c-'a');
            default: printf("%c\n",c);
        }
        c=s[++i];
    }
}

```

(8)

```

main()
{
    int a=5,b=6,i=0,j=0;
    switch(a)
    {
        case 5: switch(b)
            {
                case 5: i++;
                    break;
                case 6: j++;
                    break;
                default: i++;
                    j++;
            }
    }
}

```

```

        case 6: i -- +;
                j -- +;
                break;
        default: i ++ +;
                j ++ +;
    }
    printf("%d,%d\n",i,j);
}

```

(9)

```

main()
{
    int x=5;
    do {
        switch(x%2)
        {
            case 1: x -- --;
                    break;
            case 0: x ++ +;
                    break;
        }
        x -- --;
        printf("%d\n",x);
    }while(x>0);
}

```

(10)

```

main()
{
    int i,j,a[8][8];
    a[0][0]=1;
    for(i=1;i<8;i++)
    {
        a[i][0]=1;
        a[i][i]=1;
        for(j=1;j<i;j++)
            a[i][j]=a[i-1][j]-1+a[i-1][j];
    }
    for(i=0;i<8;i++)
    {
        for(j=0;j<=i;j++)
            printf("%5d",a[i][j]);
        printf("\n");
    }
}

```

4. 编写程序,并上机验证。

- (1) 求100之内的自然数中奇数之和。
- (2) 求100之内的自然数中最大的被13整除的数。
- (3) 求输入两个正整数的最大公约数和最小公倍数。

(4) 求下列分数序列的前15项之和。

$$\frac{2}{1}, \frac{3}{2}, \frac{5}{3}, \frac{8}{5}, \frac{13}{8}, \frac{21}{13}, \dots$$

(5) 求 $\sum_{n=1}^{10} n!$ (即求 $1! + 2! + \dots + 10!$ 之和)。

(6) 百鸡百钱问题。用100钱买100只鸡,公鸡一只5钱,母鸡一只3钱,雏鸡3只1钱。问共有多少种买法。

(7) 用迭代方法求 $x = \sqrt{a}$ 。迭代公式如下所示:

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right)$$

要求前后两次求出的 x 的差的绝对值小于 10^{-5} 。

(8) 求出1~1000之间的完全平方数。所谓完全平方数是指能够表示成另一个整数的平方的整数。要求每行输出8个完全平方数。

(9) 输入4个 int 型数,要求按大小顺序输出。

(10) 判断某一年是否闰年。闰年是指能被4整除,但能被100整除的不是闰年,而被400整除的是闰年。

(11) 求 $ax^2 + bx + c = 0$ 方程的解。

讨论如下情况:

① $b^2 - 4ac = 0$, 有两个相等实根。

② $b^2 - 4ac > 0$, 有两个不等实根。

③ $b^2 - 4ac < 0$, 有两个共轭复根。

④ $a = 0$, 不是二次方程。

(12) 有一函数:
$$y = \begin{cases} x & (x < 1) \\ x + 5 & (1 \leq x < 10) \\ x - 5 & (x \geq 10) \end{cases}$$

输入 x , 输出 y 值。

5. 分析下列程序的输出结果,并上机验证。

```
#include <stdio.h>
char input[] = "SSSWILTECH1\1\11W\1WALLMP1";
main()
{
    int i, c;
    for (i = 2; (c = input[i]) != '\0'; i++)
    {
        switch (c)
        {
            case 'a': putchar('i');
                    continue;
            case '1': break;
            case 1: while ((c = input[++i]) != '\1' && c != '\0');
            case 9: putchar('S');
            case 'E':
            case 'L': continue;
            default: putchar(c);
                    continue;
        }
        putchar('_');
    }
    putchar('\n');
}
```

第五章 函数和存储类

本章讲述关于函数方面的内容,包括函数的定义格式和说明方法,函数的形参和实参,函数的调用方式及返回值等。函数是 C 语言中重要概念,它是结构化程序设计的基本模块。本章还讲述关于变量和函数的存储类以及变量的作用域等内容。

5.1 函数的定义和说明

函数的定义和说明是两回事。定义函数是指按函数定义的格式,书写若干实现该函数功能的语句,通过调用该函数完成该函数的功能。函数的说明是指在函数被定义好后,在调用之前对函数的类型及参数的类型进行说明,有些情况下,可以不说明。下面讲述函数的定义格式和函数的说明方法。

5.1.1 函数的定义

如果想调用一个函数完成某种功能,必须先按其功能来定义该函数。函数定义的格式如下所示:

```
<存储类说明><数据类型说明><函数名>(<参数表>)  
<参数说明>  
{  
    <函数体>  
}
```

函数的定义可分两大部分:函数头和函数体。函数头包含<函数名>和<参数表>以及关于该函数的<存储类说明>、<数据类型说明>,还有<参数说明>。函数体是由一对花括号括起来的若干语句组成的,函数体内可以有一条语句或多条语句,也可以有复合语句,还可以是空,即该函数什么操作也不做,这是一个最简单的函数。<函数名>的起法同标识符,能够通过函数名来标明该函数的主要功能为最好。<函数名>后面跟一对圆括号,内有一个<参数表>,该<参数表>由一个或多个参数构成,多个参数之间用逗号分隔,也可以没有参数,但圆括号不可省略。定义函数时要对该函数的存储类和数据类型进行说明。函数的存储类有两种:外部函数和内部函数。外部函数用 extern 关键字加在函数名前面进行说明,常常省略,凡是不加存储类说明的函数都是外部函数。内部函数必须在函数名前面加 static 关键字进行说明,内部函数又称静态函数。函数的数据类型是该函数返回值的类型,函数的数据类型十分丰富,一般 C 语言所允许的数据类型大多都可作为函数类型,个别的除外,如联合类型等。C 语言规定,数据类型中除了 int 型的可以不要说明外,其余各种类型都需说明。如果一个函数具有参数时,则需对参数的类型进行说明。<参数说明>一般放在函数名的下一行,有的编译系统(如 Turbo C)可以把参数说明放在参数表中。

下面是函数定义的几个例子:

```
nothing()
{
}
```

该函数名字是 `nothing`，它没有参数，它的函数体是空，该函数什么也不做。空函数可以用于调试中，它表明应该调用一个函数，但该函数尚未编好，先用一个空函数顶替。

```
void nopa()
{
    printf("OK!\n");
}
```

该函数名是 `nopa`，该函数没有参数，因此不必进行参数说明。这里，`void` 关键字用来说明该函数没有返回值。该函数的函数体内仅有一个语句。

```
float max(x, y)
float x, y;
{
    float z;
    z = x > y ? x : y;
    return(z);
}
```

该函数名是 `max`，它有两个参数 `x, y`，它们都是 `float` 型数，由于该函数有参数，因此需要参数说明：`float x, y`；该函数的数据类型为 `float` 型，这说明该函数有返回值，而返回值的类型是 `float` 型。该函数无存储类的说明，这意味着该函数的存储类为外部的。该函数体是由3条语句组成的。这里有说明语句，它被放在执行语句的前面。

在函数的定义中还应注意如下几个问题：

(1) C 语言中的函数可分为有返回值和无返回值两大类。在有返回值的函数定义中，除 `int` 型返回值外都必须对返回值的类型进行说明。在无返回值的函数定义中，可以加上无返回值的说明符 `void`，也可以不加。这样，对于一个不加数据类型说明的函数可能是无返回值的，也可能是有 `int` 型返回值的。

(2) 函数的定义不能嵌套。这就是说，不能在一个函数的函数体内再定义一个函数。例如，下列写法是错误的：

```
f1(x, y)
int x, y;
{
    :
    f2(a, b)
    int a, b;
    {
        return(a + b);
    }
    :
}
```

这种企图在 `f1()` 函数中再定义一个 `f2()` 函数的做法是错误的。但是，函数的调用允许嵌套，这就是说，可以在一个函数体内调用另一个函数，也可以进行自身调用（又称递归调用）。

(3) 如果一个函数的函数体中有变量的定义或说明时,一定要放在执行语句的前边,不可放在中间或后面,否则要出编译错。

5.1.2 函数的说明

函数定义好后,在调用之前一般地要进行说明。函数的说明方法有如下两种:

一是只说明函数的类型,这称为简单说明。例如,对前面定义过的函数 `max()` 说明如下:

```
float max();
```

这种说明可以单独一行,也可以与其他同类型的变量放在一起。

二是不仅说明函数的类型还要说明其参数的类型,这称为原型说明。例如,对前面已定义过的函数 `max()` 原型说明如下:

```
float max (float, float);
```

原型说明要比简单说明复杂一些,但是用原型说明后,在函数调用时系统将其参数的类型进行检查,如果发现不一致,则报错。如果用简单说明方法,则在调用时不作参数类型是否一致的检查,即使不一致也不报错。由此可见,原型说明比简单说明要安全些。

在实际使用中关于函数的说明还需注意如下几点:

(1) 在下列情况下,函数在调用之前不必说明:在定义函数时没有加任何说明;或者是该函数无返回值又没有加 `void` 说明符;或者是该函数具有 `int` 型返回值而省略说明符。

(2) 在下列情况下,函数在调用之前必须说明,如不说明,将出现编译错:即在定义函数时加了说明符,包含 `void` 在内,又是先调用后定义(即调用在定义之前),则调用之前必须说明。

(3) 在下列情况下,即在定义函数时加了说明符,包含 `void` 在内,而先定义后调用(即定义在调用之前),则调用函数之前可以说明也可以不说明。

有时为了使得系统能够检查函数调用时其参数类型是否一致,对于不必须说明函数的情况也对函数进行说明。

5.2 函数的参数和返回值

5.2.1 函数的参数

C 语言中,有的函数有参数,可以一个或多个,多个参数用逗号分隔,而有的函数没有参数,但是函数名后面的圆括号不可省略,例如,前面讲过的主函数 `main()` 就是属于没有参数的函数。但是,主函数也可以有参数,后面会讲到。

函数参数分为形式参数(简称形参)和实在参数(简称实参)两种。形参是指在定义函数时,〈参数表〉中的参数,因为该参数在该函数被调用之前是没有确定值的,只是形式上的参数,只有被调用时通过实参来获取值。实参是指调用函数的参数,因为该参数已具有确定的值,是实在的参数。实参可以是表达式,在调用之前先计算出表达式的值。

在实际应用中关于形参和实参的使用还应注意如下几点:

(1) 在定义函数时,所指定的形参在该函数被调用之前是不被分配内存单元的。只有在发生函数调用时,才给形参分配单元,并且赋值,一旦函数调用结束后,形参所占的内存单元又被释放掉。因此,形参属于局部变量,其作用范围仅在定义它的函数体内。

(2) 函数调用时所用的实参是一个具有确定值的表达式,调用时先计算表达式的值,再将

其值传递给对应的形参。

(3) 函数的形参是属于定义它的函数的局部变量。因此,允许一个函数的形参和实参同名,因为它们在内存中占有不同的存储单元。

(4) 函数调用要求形参和实参在个数上相等,并且对应的参数类型相同。否则将会发生“个数不匹配”或“类型不一致”的错误。一般情况下,即使不报错误信息,也会造成结果不正确。

下面举一个简单的函数调用的例子,通过它进一步说明函数的定义格式,函数的说明方法和形参与实参的不同。

[例5.1] 求两个 float 型数的和。

程序内容如下:

```
main()
{
    float a,b,sum;
    float fadd();
    a=5.6;
    b=7.2;
    sum=fadd(a,b);
    printf("%.2f\n",sum);
}

float fadd(x,y)
float x,y;
{
    return(x+y);
}
```

执行该程序输出如下结果:

12.80

说明:

(1) 该程序中定义了两个函数,main()和 fadd()。前一个是主函数,它无参数,也无返回值。后一个是具有 float 型返回值的带有参数的函数,该函数有2个参数 x 和 y,它们都是 float 型变量。

(2) 在主函数 main()中,float fadd();这是对函数 fadd()的简单说明,只说明函数的类型是 float 型(即返回值的类型)。如果用原型说明应采用如下格式:

```
float fadd(float, float);
```

这种情况下,对 fadd()函数必须说明,因为该函数返回值非 int 型的,并且又是调用在先定义在后。如不说明,则会出现编译错,读者可以上机验证。

(3) 该程序中,x 和 y 是 fadd()函数的两个形参,关于形参类型说明的方法,除了本程序中说明方法外,还可以采用如下方法:

```
float fadd(float x, float y)
```

即在参数表中就说明了参数的类型。但是,这种说明方法并不是所有编译系统都允许。而 Turbo C 编译系统是允许的。

该程序中,a 和 b 是 fadd()函数的两个实参,在调用该函数时,a 和 b 的值是确定的。

该程序中的函数调用时满足了形参与实参个数相等,对应类型相同的要求。

5.2.2 函数的返回值

C语言中,有的函数带有返回值,有的函数不带有返回值。

函数的返回值都是通过 `return(<表达式>)` 语句来实现的。例如,在本章例5.1中, `fadd()` 函数的函数体中的 `return(x+y);` 语句就是将表达式 `x+y` 的值作为返回值,返回给调用函数 `fadd()` 的值赋给变量 `sum`。返回值的类型是该函数的类型。带有返回值的 `return` 语句具体实现过程如下:

- (1) 先计算 `return` 语句中 `<表达式>` 的值;
- (2) 根据函数的类型对 `<表达式>` 的类型进行转换,使得 `<表达式>` 的类型转换成为函数的类型;
- (3) 将 `<表达式>` 的值和类型返回给调用函数,作为调用函数的值和类型。一般情况下,要设置一个变量来接收这一返回值。
- (4) 将程序流的控制权交给调用函数,继续执行调用函数下边的语句。

在不带返回值的 `return` 语句中,不执行上述(1)至(3)步骤,只执行(4)步骤。

函数的返回值(简称函数值)的类型是在定义函数时指出的。一般情况下,除了返回值为 `int` 型的不要说明函数的类型外,对非 `int` 型返回值的函数都应指出函数值的类型。函数值的类型和 `return` 语句中表达式的类型可能不一致,这时需要转换,即将表达式的类型转换成为函数值的类型,这种转换是自动进行的。下面通过一个实例说明当函数值的类型与 `return` 语句表达式的类型不一致的情况时是如何自动转换的。

[例5.2] 求两个输入的浮点数的和,最后的和用 `int` 型表示。

程序内容如下:

```
main()
{
    float x,y;
    int sum;
    printf("Input x,y: ");
    scanf("%f%f",&x,&y);
    sum=add(x,y);
    printf("sum=%d\n",sum);
}
add(x,y)
float x,y;
{
    return(x+y);
}
```

运行情况如下:

```
Input x, y: 1.3 7.9
sum=9
```

说明:

- (1) 该程序由两个函数组成,一个是 `main()` 函数,它无参数,也无返回值;另一个是 `add()` 函数,它有两个参数 `x` 和 `y`,都是 `float` 型的。它有 `int` 型返回值。因此,定义该函数时可以不加

类型说明,调用该函数之前也可不加说明。

(2) 在 `main()` 中, `x` 和 `y` 作为 `add()` 函数的实参;在 `add()` 函数中, `x` 和 `y` 作为 `add` 函数的形参。虽然名字相同,但它们在内存中有着不同的存储单元。调用 `add()` 函数时,其形参与实参是一致的,即个数相等,对应类型相同。

(3) 在被调用函数 `add()` 中,返回值的表达式 `x+y` 的类型是 `float` 型的,其值应为 `1.3+7.9`,即为 `9.2`。由于函数 `add()` 的类型为 `int` 型的,因此,要将 `9.2` 自动转换为 `int` 型数 `9`,再返回给调用函数,即赋值给 `int` 型变量 `sum`。

5.3 函数的调用

C 语言中,函数之间通过调用联系起来,函数的调用基本上是属于传值调用。所谓传值调用是将调用函数的实参值传递给被调用函数的形参,使得被调用函数的形参获得值。传值调用中,可以传递变量的值,也可以传递变量的地址值,这是两种不同的传值方式。为了将这两种传值方式加以区别,我们称传递变量值的调用为传值调用,称传递变量地址值的调用为传址调用。这两种调用方式实质都是传值调用,都遵循传值调用的特点:要求实参与形参个数相等,对应的类型相同。下面分别讲述这两种不同方式的传值调用的各自的特点。另外,本节还将讲述函数的嵌套调用和函数的递归调用的方法。

5.3.1 传值调用的特点

这里所讨论的传值调用是指传递变量值的调用方式。在这种调用方式中,实参使用变量名或者表达式,形参使用变量名。在调用时,调用函数将实参值拷贝到一个副本给形参,即使形参按顺序从对应的实参中获得值,这就相当于将实参值对应地赋给形参,使形参获值。这种调用方式具有如下特点:被调用函数的参数值被改变而不会影响调用函数的参数值,因此安全性好。

[例5.3] 分析下列程序输出结果。

```
main()
{
    float a,b;
    void f1();
    a=7.2;
    b=3.6;
    f1(a,b);
    printf("%.2f,%.2f\n",a,b);
}

void f1(x,y)
float x,y;
{
    x+=0.5;
    y-=0.5;
    printf("%.2f,%.2f\n",x,y);
}
```

执行该程序输出结果如下：

7.70, 3.10

7.20, 3.60

说明：

(1) 该程序由两个函数组成，一个是主函数 `main()`，另一个是被 `main()` 函数调用的被调用函数 `f1()`。在定义 `f1()` 函数时，使用了 `void` 说明该函数无返回值。因此，在调用 `f1()` 函数之前必须说明 `f1()` 函数，因为调用在先定义在后。

(2) 主函数中，调用 `f1()` 函数时，两个实参都是变量，即 `a` 和 `b`。被调用函数的两个形参也都是变量，即 `x` 和 `y`，这是属传递变量值的传值调用方式。在被调用函数 `f1()` 中，通过两个赋值表达式语句分别对形参 `x` 和 `y` 的值进行改变，并通过 `printf()` 语句输出改变后的 `x` 和 `y` 的值：7.70, 3.10。返回主函数后，执行 `printf()` 语句输出主函数中的 `a` 和 `b` 值为 7.20, 3.60。可见，`a` 和 `b` 值没有因为调用函数 `f1()` 而改变其值。尽管在 `f1()` 函数中对其形参 `x` 和 `y` 进行了改变。可见，这种传值调用方式使得被调用函数中参数值的改变不影响调用函数的参数值，其原因是因为这种调用方式是调用函数将其实参值拷贝了一个副本给了被调用函数的形参，被调用函数中改变形参的值，只是改变了副本中的值，对调用函数中的“正本”没有改变。

5.3.2 传址调用的特点

传址调用是指在调用时传递变量地址值的传值调用。传址调用时要求调用函数的实参用地址值，而被调用函数的形参用指针，于是函数之间进行地址值的传递。这种传递是将实参的变量地址值传递给形参指针，即让形参指针指向实参变量，这种传递方式与调用函数拷贝实参值的副本给形参是不同的，它是让形参指针直接指向实参的变量。这种传址调用具有如下特点：被调用函数中可以通过改变形参所指向的内容来改变调用函数的实参值。这一特点与前面讲过的传值调用是截然不同的。传址调用可以通过改变形参所指向的内容来改变实参值，这就提供了函数之间进行信息传递的又一渠道，并且这种传递信息的方式还可以克服前面讲过的返回值方式的只传一个信息的局限性。因此，传址调用在 C 语言函数调用中是经常采用的方式。后面还会看到这种调用方式将会带来其他好处。

下面举一个传值调用方式和传址调用方式不同的例子，并且将看到这两种传递方式可在同一个函数调用中实现。

[例5.4] 分析下列程序输出结果。

```
main()
{
    int a,b,c;
    a=b=c=5;
    f1(a,&b,&c);
    printf("%d,%d,%d\n",a,b,c);
}

f1(x,y,z)
int x,*y,*z;
{
    x*=2;
    *y+=x;
```

```

    *z=x+*y;
    printf("%d,%d,%d\n",x,*y,*z);
}

```

执行该程序输出如下结果:

```

10,15,25
5,15,25

```

说明:

(1) 该程序由主函数 main() 和被调用函数 f1() 组成, f1() 函数无返回值, 由于定义 f1() 时没有加任何类型说明符, 尽管调用在先定义在后也不必说明。

(2) 在调用函数 f1() 时, 3 个实参中, 一个是用变量名, 另外 2 个用变量的地址值; 在对应的形参中, 一个是用变量名, 另外 2 个用指针。在调用 f1() 函数中, 有一个参数属于传值调用, 2 个参数属于传址调用。在被调用函数 f1() 中, 改变了变量 x 的值, 由于传值调用对调用函数对应实参 a 的值并没有被改变, 这是传值调用的特点。同时, 通过用运算符 * 对指针 y 和 z 取其内容, 并且通过赋值表达式语句来改变了 x 和 y 的内容, 于是调用函数中对应的实参 b 和 c 的值被相应改变, 这便是传址调用的特点。

该例程序中值得注意的是要在传址调用中通过被调用函数来改变调用函数的参数值时, 一定要改变形参指针所指向的内容, 而不能只改变形参的地址值, 改变形参地址值是不会影响对应实参的变量值的。下面的例子将说明这一点。

[例 5.5] 分析下列程序的输出结果。

```

main()
{
    int a,b,c,d;
    a=b=c=d=5;
    f1(&a,&b);
    f2(&c,&d);
    printf("%d,%d,%d,%d\n",a,b,c,d);
}

f1(x,y)
int *x,*y;
{
    *x*=2;
    *y+=*x;
    printf("%d,%d\n",*x,*y);
}

f2(m,n)
int *m,*n;
{
    int p,q;
    p=q=8;
    m=&p;
    n=&q;
    printf("%d,%d\n",*m,*n);
}

```

执行该程序输出结果如下：

```
10,15
8,8
10,15,5,5
```

说明：

(1) 该程序由三个函数组成，除了主函数 `main()` 外，还有二个被调用函数 `f1()` 和 `f2()`。这两个被调用函数都是没有返回值的。在 `main()` 函数中，先调用 `f1()` 函数，再调用 `f2()` 函数。

(2) 在调用 `f1()` 函数时采用了传址调用方式，即实参用变量 `a` 和 `b` 的地址值 `&a` 和 `&b`，形参用指针 `x` 和 `y`。在 `f1()` 函数中，通过使用取内容运算符改变了指针 `x` 和 `y` 所指向的变量的值（即改变了指针所指向的内容）。因此，`f1()` 函数被调用后，主函数中变量 `a` 和 `b` 的值发生了变化，这是通过传址调用改变调用函数参数值的又一个例子。

(3) 在调用 `f2()` 函数时也采用了传址调用方式，即实参用变量 `c` 和 `d` 的地址值 `&c` 和 `&d`，形参用指针 `m` 和 `n`。在 `f2()` 函数中，形参 `m` 和 `n` 的地址值发生了变化，使 `m` 和 `n` 指向了变量 `p` 和 `q`，于是 `m` 和 `n` 所指向的内容将是 `p` 和 `q` 变量的值 8。这时，如果要再改变 `m` 和 `n` 所指向的内容，只是改变变量 `p` 和 `q` 的值，也不会影响调用函数的参数值。因此，简单地说“传址调用中通过改变形参的值来改变调用函数中参数的值”这句话是不确切的。一定要指出在被调用函数中通过改变形参所指的内容才能改变调用函数中实参的值。

5.3.3 数组名作参数的函数调用

数组元素可作为实参，实现传值调用，这与变量名作实参一样，都是单向传递的。

数组名作函数实参与数组元素作实参是不同的，因为 C 语言规定数组名是一个地址值，即是该数组首元素的地址值。因此，数组名作实参时，要求形参也是数组名或者指向数组的指针，关于指向数组的指针将在下一章“指针”中讲解，这时实现的不是传值调用，而是传址调用。调用函数不是将整个数组的所有元素拷贝成副本传递给被调用函数，而是只将其数组的首元素地址值传给形参数组，于是这两个数组将共同占用同一段内存单元，这就是让形参数组的首元素地址与实参的首元素地址相同，使得这两个数组的对应元素同占一个内存单元。因此要求这两个数组要类型相同，数组的大小可以一致，也可以不一致。如果要使形参数组得到实参数组的全部元素，则形参数组与实参数组应大小一致。

下面举例说明用数组名作函数参数的调用方法。

[例5.6] 编程对某一数组中的各个 `int` 型数进行由小到大的排序。

本例中采用算法简单的选择排序法。

程序内容如下：

```
main()
{
    void sort();
    static int a[8]={5,-2,9,87,0,6,21,49};
    int i,n;
    n=8;
    sort(a,n);
    for(i=0;i<8;i++)
```

```

        printf("%4d",a[i]);
    printf("\n");
}
void sort(b,m)
int b[],m;
{
    int i,j,k,temp;
    for(i=0;i<m-1;i++)
    {
        k=i;
        for(j=i+1;j<m;j++)
            if(b[j]<b[k])
                k=j;
        temp=b[k];
        b[k]=b[i];
        b[i]=temp;
    }
}

```

执行该程序输出结果如下：

-2 0 5 6 9 21 49 87

说明：

(1) 该程序是通过调用 sort() 函数对已知数组 a 中的 8 个 int 型数进行由小到大排序。被调用函数 sort() 是没有返回值的。

(2) 主函数中调用函数格式如下：

sort(a, n)

其中, 实参 a 是一个已知的数组名, 该数组是一个一维数组, 8 个 int 型元素。实参 n 的值为 8。可见, sort() 函数的两个实参中, 一个是地址值, 另一个变量值。

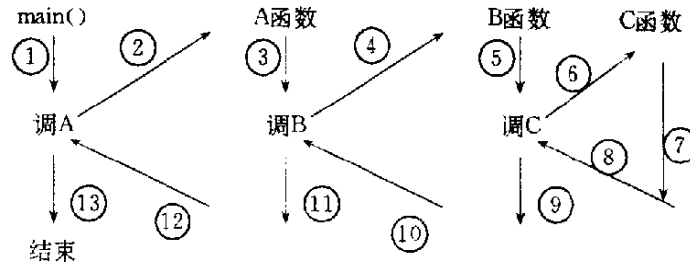
被调用函数 sort() 的两个形参中, 一个是数组名 b, 它是一个没有指定大小的一维 int 型数组, 其类型与数组 a 相同, 其大小将与所对应的实参数组大小相等。另一个形参是变量 m。因此, 不难看出该函数调用中, 第一个参数是属于传址调用, 第二个参数是传值调用。在传址调用中, 将数组 a 的首元素地址传给 b 数组, 使得 a、b 两个数组共占同一段内存单元。由于是传址调用不必拷贝数组元素的副本, 因此, 效率较高, 节省了时间和空间的开销。这是传址调用的又一好处。

由于 a、b 数组共占同一段内存单元, 因此在 sort() 函数中, 通过 b 数组元素的改变, 而使得数组 a 的元素发生了变化, 于是完成由小到大的排序。

(3) 在函数 sort() 中, 选择排序法的算法是这样的: 通过双重 for 循环, 每作一次外重 for 循环, 从指定的数中挑出最小的一个放在指定的元素位置, 例如, 第一次外重 for 循环, 从 8 个数中挑出最小的放在第 0 个元素的位置, 第二次外重循环从剩下的 7 个数中再挑出最小的放在第 1 个元素的位置, 依次类推, 外重 for 循环共进行 7 次, 便将 8 个数的顺序由小到大排好。内重 for 循环是用来确定在要查找的数中找出最小数所对应的下标值, 并将它赋给 k 变量。然后, 通过三个赋值语句将数值最小的那个元素换到待选数中的最前边, 即下标值为 i 的位置。这种排序方法每次找出一个最小的数放在前边, 直到最后剩下一个数为止。

5.3.4 函数的嵌套调用

所谓函数的嵌套调用是指在调用一个函数的过程中,又调用另外一个函数。例如,在调用 A 函数的过程中,还可以调用 B 函数,在调用 B 函数的过程中,还可以调用 C 函数,……。当 C 函数调用结束后返回到 B 函数,当 B 函数调用结束后返回到 A 函数,当 A 函数调用结束后再返回到 A 的调用函数中。假定 main()调用 A 函数,上述的嵌套调用关系可用下图所示:



图中①②③…表示了执行嵌套过程的顺序号。即从①开始,经过了三级嵌套,到⑬结束。

这里还需重申一遍,C 语言中函数的定义不允许嵌套,就是说不允许在函数中定义函数,C 语言程序中若干个函数都是平行的、独立的。函数之间是通过调用联系的。函数的调用是允许嵌套的,就是说在调用某个函数的过程中,还允许调用其他函数。

下面举一个函数嵌套调用的例子,通过该例搞清调用、返回之间的关系。

[例5.7] 分析下列程序的输出结果。

```
main()
{
    printf("I'm in main. \n");
    a();
    printf("I'm finally back in main. \n");
}

a()
{
    printf("Now, I'm in a. \n");
    b();
    printf("Here, I'm back in a. \n");
}

b()
{
    printf("Now, I'm in b. \n");
    c();
    printf("Back in b. \n");
}

c()
{
    printf("And now, I'm in c. \n");
}
```

请读者分析该程序后,写出输出结果。

[例5.8] 编程求出下式之和。

$$1^k + 2^k + 3^k + \cdots + n^k$$

假定 k 为4, n 为6,编写求上述和的程序如下:

```
#define K 4
#define N 6
main()
{
    printf("Sum of %dth powers of integers from 1 to %d=",K,N);
    printf("%d\n",sum_of_powers(K,N));
}
sum_of_powers(k,n)
int k,n;
{
    int i,sum=0;
    for(i=1;i<=n;i++)
        sum+=powers(i,k);
    return(sum);
}
powers(m,n)
int m,n;
{
    int i,product=1;
    for(i=1;i<=n;i++)
        product*=m;
    return(product);
}
```

执行该程序输出结果如下:

sum of 4th powers of integers from 1 to 6=2275

说明:

(1) 该程序由 `main()`, `sum-of-powers()` 和 `powers()` 三个函数组成, `main()` 中调用 `sum-of-powers()` 函数, 该函数返回一个 `int` 型数值, 而 `sum-of-powers()` 函数中又调用 `powers()` 函数, 该函数也返回一个 `int` 型数值。从中可见, 函数之间的嵌套调用在实际编程中是经常使用的。

(2) 在主函数中, 调用 `sum-of-powers()` 函数时, 实参是两个符号常量 K 和 N 。可见符号常量与一般常量一样都可作为函数的实参。本程序中的两次函数调用都属于传值调用, 函数之间的信息传递是通过返回值来实现的。

5.3.5 函数的递归调用

C 语言中允许使用函数的递归调用, 这是 C 语言的又一特点。所谓函数的递归调用是指在调用一个函数的过程中出现直接地或间接地调用该函数自身。例如, 在调用 `f1()` 函数的过程中, 又出现了调用 `f1()` 函数, 这称为直接调用; 而在调用 `f1()` 函数过程中出现了调用 `f2()` 函数, 又在调用 `f2()` 函数过程中出现了调用 `f1()` 函数, 这称为间接调用。

1. 递归调用的特点

实际中不是所有的问题都可以采用递归调用的方法。只有满足下列要求的问题才可使用递归调用方法来解决：

能够将原有的问题化为一个新的问题，而新的问题的解决办法与原问题的解决办法相同，按这一原则依次地化分下去，最终化分出来的新的问题可以解决。

实际中有意义的递归问题都是经过有限次数的化分，最终可获得解决。这是有限递归问题，而那些无限递归问题在实际中是没有意义的。下面举一个有限递归的例子。例如，求 $5!$ 。由于 $5!$ 可以化为 $5 * 4!$ ，而 $4!$ 又可化为 $4 * 3!$ ，而 $3!$ 可化为 $3 * 2!$ ，而 $2!$ 可化为 $2 * 1!$ ，最后， $1!$ 可化为 $1 * 0!$ ，而 $0!$ 是已知的，即为1。于是，可知 $5!$ 等于 $5 * 4 * 3 * 2 * 1 * 1$ ，即120。这是一个简单的典型的递归调用的例子。

使用递归调用方法编写程序简洁清晰，可读性强。因此，人们都喜欢用递归调用的方法来解决某些问题。但是，用这种方法编写的程序执行起来在时间和空间的开销上都比较大，即要占用较多的内存单元，又要花费很多的计算时间。因为递归调用时要占用内存的许多单元存放“递推”的中间结果，较复杂的递归占用内存空间较多。因此，在一些内存小速度慢的小机器上最好不要采用递归调用的办法，不然，效率很低。一般的凡是可用递归调用方法编写的程序都可以用迭代的方法来编写。一般说来，相同的问题用迭代方法编写要比用递归调用方法编写的源程序长些。

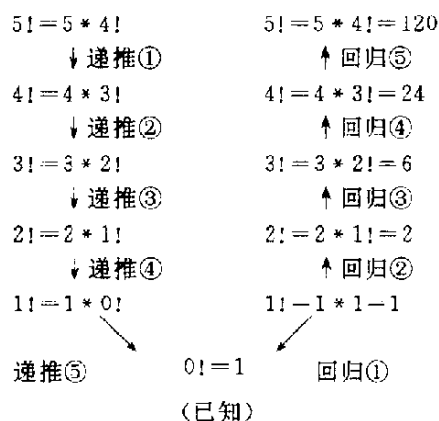
2. 递归调用的过程

递归调用的过程可分为如下两个阶段：

第一阶段称为“递推”阶段：将原问题不断化为新问题，逐渐从未知的向已知的方向推测，最终达到已知的条件，即递归结束条件。

第二阶段称为“回归”阶段：该阶段是从已知条件出发，按“递推”的逆过程，逐一求值回归，最后到递推的开始之处，完成递归调用。可见，“回归”的过程是“递推”的逆过程。

下面以求 $5!$ 为例写出递归调用的全过程如下所示：



从上述列举的递推过程中可以看出，有实际意义的递归应该是有限的，即递推若干次后，将出现已知条件，并且每递推一次都是向已知条件接近一步。这里的已知条件就是递推结束条件。上例中， $0! = 1$ 是已知条件，即递推结束条件。在回归的过程中，是按照递推的逆过程进行的，最后得到原问题的解。

[例5.9] 用递归调用方法编程求某个正整数的阶乘。

程序内容如下：

```

main()
{
    int n;
    scanf("%d",&n);
    printf("%d\n",fac(n));
}
fac(n)
int n;
{
    int p;
    if(n==0) p=1;
    else
        p=n*fac(n-1);
    return(p);
}

```

执行该程序,从键盘上输入一数后,输出结果如下:

```

5 ↵
120

```

说明:该程序由主函数 main() 和 fac() 函数组成, fac() 函数中采用了递归调用的形式,即在 fac(n) 函数中调用 fac(n-1)。该递归结束条件时, n 等于 0, p 值为 1。每递归一次 n 减 1, 经过若干次递归后, n 会为 0。

[例 5. 10] 汉诺(Hanoi)塔问题。这是一个典型的递归调用问题。该问题描述如下: 在一张桌子上有 A, B, C 三处, 在 A 处有 n 个盘子, 每个盘子大小不等, 大的在下小的在上。要求将 A 处的 n 个盘子移到 C 处, 可以借助于 B 处, 每次移动只允许动一个盘子, 在移动过程中在 A, B, C 三处都应保持大盘在下, 小盘在上。编程打印出移动的过程。

分析算法如下:

该题目使用递归调用的方法编程。将 n 个盘子从 A 处移到 C 处可分为如下三个步骤:

- (1) 将 A 处的 n-1 个盘子借助 C 处移到 B 处。
- (2) 把 A 处剩下的一个盘子移到 C 处。
- (3) 再将 B 处的 n-1 个盘子借助于 A 处移到 C 处。

这样完成了 n 个盘子的移动。

分析上述的三步操作可以发现, 第 1 步和第 3 步所采用的方法是一样的, 都是将 n-1 个盘子从某一处移到另一处, 只是移动的位置不同。因此, 可将上述的三步骤化简为如下两步骤:

- (1) 将 n-1 个盘子从一处移到另一处。
- (2) 将一个盘子从一处移动另一处。

把这两个步操作分别用两个函数来描述。第一步操作用 move_n() 函数来实现, 第二步操作用 move_1() 函数来实现。

move_n() 函数有 4 个参数, 分别为 n, a, b, c, 表示“将 n 个盘子从 A 处借助 B 处移到 C 处”。

该函数具体定义如下:

```

void move_n(n,a,b,c)
int n;
char a,b,c;
{
    if(n==1) move_1(a,c);
    else
    {
        move_n(n-1,a,c,b);
        move_1(a,c);
        move_n(n-1,b,a,c);
    }
}

```

move_1()函数有2个参数,一个是 from,另一个是 to,表示将1个盘子从 from 处移到 to 处。该函数具体定义如下:

```

int m;
void move_1(from,to)
char from,to;
{
    if (m%8==0)
        printf("\n");
    printf(" %c=>%c ",from,to);
    m++;
}

```

该程序主函数内容如下:

```

void move_1(),move_n();
main()
{
    int d;
    printf("Input the number of disks: ");
    scanf("%d",&d);
    printf("The steps to moving %4d disks:\n");
    move_n(d,'A','B','C');
    printf("\n");
}

```

执行该程序输出如下信息:

Input the number of disks: 3 ↙

The steps to moving 3 disks:

A⇒C A⇒B C⇒B A⇒C B⇒A B⇒C A⇒C

再运行一次该程序:

Input the number of disks: 4 ↙

The steps to moving 4 disks:

A⇒B A⇒C B⇒C A⇒B C⇒A C⇒B A⇒B A⇒C

B⇒C B⇒A C⇒A B⇒C A⇒B A⇒C B⇒C

再运行一次该程序:

Input the number of disks: 5 ✓

The steps to moving 5 disks:

A⇒C A⇒B C⇒B A⇒C B⇒A B⇒C A⇒C A⇒B
C⇒B C⇒A B⇒A C⇒B A⇒C A⇒B C⇒B A⇒C
B⇒A B⇒C A⇒C B⇒A C⇒B C⇒A B⇒A B⇒C
A⇒C A⇒B C⇒B A⇒C B⇒A B⇒C A⇒C

说明:

(1) 该程序由3个函数组成: `main()`, `move_1()`和 `move_n()`。主函数 `main()`中调用 `move_n()`函数,在 `move_n()`函数中又调用 `move_1()`函数。由于 `move_1()`函数和 `move_n()`函数在定义时使用了 `void` 来说明它们是无返回值的,因此,如果 `main()`在前,`move_n()`和 `move_1()`函数在后,则调用后两个函数之前要说明,该程序将说明放在文件头,即主函数的前面。

(2) `move_n()`函数采用了递归调用方法,即在 `move_n(n, ...)`函数体中两次调用 `move_n(n-1, ...)`函数。该问题采用递归调用方法编程是很简洁明了的。用其他编程将是十分复杂的。

(3) 本例中三次运行这个程序,给出了输入分别为3个盘子、4个盘和5个盘子的三组输出数据,表明了三种移动的过程。

5.4 作用域规则

本节讨论标识符的作用域规则和重新定义的同名变量的作用域规定。

5.4.1 标识符的作用域规则

标识符的作用域规则描述如下:

标识符只能在说明它或定义它的函数体或分程序内是可见的,而在该函数体或分程序外则是不可见的。

现将这段描述说明如下:

(1) 大多数的标识符对它说明或对它定义是一回事,只有少数的标识符对它说明或对它定义是两回事,例如,外部变量和函数。

(2) 标识符包含了变量、函数、语句标号、符号常量等名字,凡是用标识符规则定义的各种单词都属于标识符。

(3) 可见的指的是可以进行存取、访问,可见时对它的改变是有效的,不可见的是指不能对它进行存取、访问及其他操作。可见的与存在的是两个不同概念:可见的是指在其作用域内,对它可以操作;存在的是指其寿命,仍存放在内存内,没有被释放。可见的一定是存在的,不存在一定不可见,但是存在的都不一定都可见,有的标识符它虽然存在,但不可见。后面在存储类中会详细讲到,内部静态变量有时不可见了,但它仍然存在。

各种标识符的作用域是不相同的,有的是程序级,有的是文件级,还有函数级和程序段级。下面列举出一些标识符的不同作用域级别。

函数有程序级的,如外部函数;有文件级的,如内部函数。

变量中,外部变量属于程序级,外部静态变量属于文件级,函数形参属于函数级,还有定义或说明在函数内的自动变量和内部静态变量以及寄存器变量都属于函数级的,定义在分程序内的自动的、内部静态的和寄存器的变量属于程序段级的。

语句标号属于函数级的。

符号常量属于文件级的。

这里所说的程序级的是指作用范围(即作用域)在整个程序内,包含该程序的所用文件。同样,文件级的是指作用范围在定义它的文件中,往往是从定义时开始。函数级的是指作用范围在定义它的函数体内;程序段级的是指作用域在定义它的分程序内。

5.4.2 重新定义变量的作用域规定

C语言中,一般说来变量不能重复定义,这是指在相同的作用域内,不能有同名变量存在。但是,在不同的作用域内,允许对某个变量进行重新定义。例如,在某个函数体内定义了一个int型变量a,这时不可以在同一个函数体内再定义一个float型变量a。但是,可以在该函数体内的某个分程序中,对变量a重新定义,于是在整个函数体内的不同分程序中将出现同名变量。这时,它们的作用域又是如何规定的呢?关于重新定义的变量的作用域规定如下:

在函数体或分程序内可以各自定义变量,在函数体内又允许嵌套分程序。在函数体或外层分程序中定义的变量,如果在内层分程序中没有重新定义,则在内层分程序中将仍然有效;如果在内层分程序中进行了重新定义,则外层中的该变量被隐藏起来,而在内层中起作用的是内层重新定义的变量,当退出内层程序又回到外层分程序或函数体内时,外层定义的该变量又恢复出来,仍然起作用。

下面通过一个程序例子来理解和体会上述规定的含义。

[例5.11] 分析下列程序的输出结果。

```
main()
{
    int a=2,b=4,c=6;
    printf("%d,%d,%d\n",a,b,c);
    {
        int b=8;
        float c=8.8;
        printf("%d,%c,%.1f\n",a,b,c);
        a=b;
        {
            int c;
            c=b;
            printf("%d,%d,%d\n",a,b,c);
        }
        printf("%d,%d,%.1f\n",a,b,c);
    }
    printf("%d,%d,%d\n",a,b,c);
}
```

执行该程序输出结果如下:

```
2,4,6
2,8,8.8
8,8,8
8,8,8.8
8,4,6
```

说明:该程序仅有一个主函数 `main()`,该函数中定义了 `a, b, c` 三个 `int` 型变量,并赋了初值。函数体内包含了一个分程序。该分程序内,重新定义了变量 `b` 和 `c`,并赋了初值,这时函数体内定义的 `b` 和 `c` 在这里被隐藏起来,而重新定义的 `b` 和 `c` 在起作用,这时输出的 `a` 值是原来的, `b` 和 `c` 是重新定义的。该分程序内,对 `a` 重新赋值,这不是重新定义,只是改变了 `a` 的值,因为 `a` 在这里是可见的。然后,在该分程序中,又定义了一个分程序,称内层分程序。在内层分程序中,重新定义了 `c`,并给它赋了值,在这里,开始定义的 `a` 仍然可见,开始定义的 `b` 仍被隐藏,外层分程序中重新定义的 `b` 是可见的,这里定义的 `c` 在起作用,外层分程序定义的 `c` 也被隐藏起来。当退出内层分程序后,开始定义的 `a` 和外层分程序定义的 `b` 和 `c` 是可见的。这时在内层分程序中,被隐藏的 `b` 和 `c` 被恢复了。当退出外层分程序后,开始定义的 `a` 和 `b, c` 都可见的。而被重新定义的 `b` 和 `c` 不可见了。值得注意的是变量 `a` 在整个函数体的任何部分都是可见的,因此,不论在何处它的值被改变后,则仍然保留其改变后的值。

5.5 存储类

本节内容主要是讲述变量的存储类和函数的存储类。先解释一下存储类的概念。存储类是指数据被存放的地方不同而进行的分类。C 语言中,有的变量可以被存放在 CPU 的通用寄存器中,多数的数据是被存放在内存中。存放在内存中的数据又被分为静态存储区和动态存储区两种,两种不同存储区内分别存放不同种类存储类变量的数据。因此,我们说,变量按其作用域可分为全局变量和局部变量。按其寿命可分为静态存储变量和动态存储变量。静态存储变量是指那些数据被存放在静态存储区的变量,动态存储变量是指那些数据被存放在动态存储区的变量。

5.5.1 变量的存储类

C 语言规定变量的存储类分为如下4种:

- (1) 自动存储类变量(`auto`);
- (2) 寄存器存储类变量(`register`);
- (3) 外部类变量(`extern`);
- (4) 静态类变量(`static`)。

下面从两个方面来讨论不同存储类的特征,一是作用域和寿命,二是初始化。

1. 不同存储类变量的作用域和寿命

(1) 自动类变量的作用域是在定义它的函数体或分程序内,一旦退出了该函数体或分程序,则是不可见的。这类变量的寿命是短的,它被存放在内存的动态存储区内。每次进入定义它的分程序或函数体内被动态分配存储区域,一旦退出该分程序或函数体后,所占用的内存区域被释放掉,即不存在了。

自动类变量一定出现在函数体或分程序内,自动类变量的存储类说明符是 `auto`。多数情况下该说明符被省略。前面讲过的程序中凡在函数体或分程序内出现的没有加存储类说明符的都是自动类的。

总之,自动类的特点是作用域小、寿命短,可见性和存在性是一致的。即可见时即存在,一旦不可见了,也就不存在了。

(2) 寄存器类变量的作用域和寿命与自动类变量相同,即作用域是在定义它的函数体或分程序内,寿命是短的。这类变量与自动类变量的区别在于寄存器类变量有可能被存放在 CPU 的通用寄存器中。如果这类变量数据被存放到通用寄存器中,则将大大提高对数据的存取速度。到底所定义的寄存器类变量能否被存放到通用寄存器中,这取决于当时 CPU 是否有空闲的通用寄存器。如果没有被存放到通用寄存器中,则按自动类变量处理。在一个程序中,定义寄存器类变量时,应注意如下几点:

① 定义的寄存器变量的个数不能太多,因为空闲的通用寄存器数目是很有限的。

② 由于通用寄存器的数据长度的限制,一般定义为寄存器变量的数据类型为 `char` 型或 `int` 型。数据长度太大的数据通用寄存器放不下。

③ 通常是选择那些使用频度较高的变量被定义为寄存器类,这样将有利于提高效率。例如,在多重循环的程序中,选择最内重循环的循环变量定义为寄存器类变量,因为它使用频度高。

(3) 外部类变量的作用域是整个程序,包括组成该程序的所有文件。在一个多文件组成的程序中,在某个文件内定义了外部类变量,它在任何一个文件中都是可见的。因此,外部类变量的作用域最大。外部变量的寿命是长的,因为外部变量被存放在内存的静态存储区。某个程序的外部变量的寿命直到该程序结束才被释放。因此,可以说外部类变量的可见性与存在性也是一致的,这就是外部类变量作用域大,寿命也长,这是外部类变量的特点。由于这一特点决定了外部类变量可以作为函数之间传递信息的一种方式。这种传递方式很方便,只要在程序中定义一个外部变量,它无论在该程序中的哪个函数内被修改,则被修改的值要一直保持到其他函数中,将改变值传给另一函数。但是,函数之间使用外部类变量传递信息不够安全。一旦产生误修改,则会造成不良后果。因此,在实际编程中尽量少用外部类变量。回顾讲过的函数之间传递信息有如下三种方式:

① 返回值方式,安全可靠,但只能返回一个值。

② 传址调用方式,既安全又可传递多个值。

③ 外部类变量方式,虽可传递多个值,但是不安全。

(4) 静态类变量又分为内部静态类和外部静态类两种。内部静态类的作用域同于自动类,即在定义它的函数或分程序内是可见的。该类变量的寿命是长的,同于外部类,即这类变量被存放在内存的静态存储区。这类变量被定义后,它将一直存在到程序结束才被释放。外部静态类变量的作用域是在定义它的文件中,并且从定义时开始。可见,这类变量的作用域是介乎于外部类和自动类之间。它的寿命同内部静态类,是长的。综观静态类变量可以看出,它的作用域和寿命,即可见性和存在性是不一致的,这便是静态类变量的特点。这就说,这类变量作用域并不是很大,但是它的寿命却是长的。虽然这类变量在它的非作用域内是不可见的,但它们却是存在的。比如,内部静态类变量当退出定义它的函数体或分程序后,便是不可见的,但是它却是存在的,它所占用的内存空间不被释放,所保存的数据也不被改变,直到程序结束才被释放。内

部静态类和外部静态类的区别仅在于作用域上,前者小些,后者大些。

2. 不同存储类变量的定义或说明方法

除了外部类变量的定义和说明不是一回事外,其他存储类变量的定义和说明是一回事。

(1) 自动类变量的定义或说明方法

定义自动类变量应在函数体或分程序内,定义时可以在变量名前加说明符 `auto`,多数情况下是 `auto` 省略。凡是在函数体或分程序内定义的变量,前边加 `auto` 或不加 `auto` 的一律为自动类变量。

(2) 寄存器类变量的定义或说明方法

定义寄存器类变量时在变量名前一定要加说明符 `register`。凡是在程序中看到变量名前面加有 `register` 的一律为寄存器类变量。

(3) 静态变量的定义或说明方法

定义静态类变量时在变量名前一定要加说明符 `static`。内部静态类变量定义在函数体或分程序内;外部静态类变量定义在函数体外,即在某个文件中,在文件首或文件尾都可以,也可以在文件中间,但是它的作用域是从定义时起。可见,内部静态类变量和外部静态类变量从定义方式上的区别仅在于是在函数体内定义还是在函数体外定义。

(4) 外部类变量的定义和说明方法

外部类变量的定义和说明是两回事。外部类变量的定义方法是在函数体外不加存储类说明符,这便是外部类变量。自动类变量的定义也可不加存储类说明符 `auto`,但它与外部类变量的区别在于外部类变量在函数体外,自动类变量在函数体内。

外部类变量在引用前一般要进行说明。说明的方法是在变量名前面加上说明符 `extern`。这种说明可以放在函数体内,也可放在函数体外,如文件开始,只要在引用之前说明即可。有时引用外部类变量也可以不说明,但是在下面两种情况下引用外部类变量必须说明:

① 在一个多文件的程序中,某个文件中定义了外部类变量,如果要在另一个文件中引用,则引用前必须说明。

② 在一个文件中,如果是先引用外部类变量,后定义外部类变量,则引用前必须说明。因为外部类变量只要求定义在函数体外,可以定义在文件头,也可以定义在文件尾,还可以定在文件中间。对于那些定义在后,引用在前的外部类变量,则需要说明。

值得注意的是对外部类变量说明时用 `extern` 说明符,定义时不用存储类说明符,只要求定义在函数体外。对一个外部类变量,只能定义一次,但可说明多次。

3. 不同存储类变量的初始化

所谓初始化是指在定义变量时赋初值。

对于一般变量定义时都可以赋初值。对自动类和寄存器类变量定义后在没有赋初值或赋值之前,其值不可使用,因为它无意义的数据。对外部类和静态类变量定义后在没有赋初值或赋值之前,它们有默认值,对 `int` 型变量为0,对浮点型变量为0.0,对 `char` 型变量为空。对于外部类或静态类变量赋初值是在编译时进行的。而自动类变量是在运行中赋初值。

对于数组只有外部类和静态类的才可以赋初值。一般编译系统不能给自动类数组赋初值。寄存器类的没有数组。

4. 全局变量和局部变量

变量从作用域来区分可分为全局变量和局部变量。

全局变量是指作用域在文件级以上的,它包含有外部类变量和外部静态类变量。

局部变量是指作用域在函数级以下的,它包含有自动类变量、寄存器类变量和内部静态类变量。另外,函数的形参也属于局部变量,因为函数形参的作用范围只在函数体内。

5. 不同存储类变量特性的小结

将不同存储类变量的特性总结如下:

存储类别	作用域	寿命	一致性	初始化	其他
自动类	在定义它的函数体或分程序内	短的	一致的	变量:没有被赋值或赋初值时无意义 数组:不能被赋初值	安全性强
寄存器类					存取速度快
外部类	整个程序的所有文件内	长的	不一致	变量:没有赋值或赋初值时有默认值 数组:可以赋初值	安全性差
静态类	内部				介乎于自动类与外部类之间
	外部				

下列通过一些例子进一步说明各种不同存储类变量的定义方法和特性。

[例5.12] 局部变量的定义和应用。

分析下列程序的输出结果。

```
main()
{
    int a;
    register int b;
    static int c;
    a=5;
    b=7;
    printf("a=%d,b=%d,c=%d\n",a,b,c);
    other();
    other();
    printf("a=%d,b=%d,c=%d\n",a,b,c);
}

other()
{
    int a=5;
    static int b=10;
    a+=10;
    b+=20;
    printf("a=%d,b=%d\n",a,b);
}
```

执行该程序输出结果如下:

```
a=5, b=7, c=0
a=15, b=30
a=15, b=50
a=5, b=7, c=0
```

说明:

(1) 在该程序的主函数 main() 中定义了自动类变量 a, 寄存器类变量 b 和内部静态类变量 c。对 a 和 b 分别赋了值, 对 c 没有赋值, 用 printf() 语句输出变量 a, b, c 值时, c 变量使用的

是默认值0。读者可以去掉给 a 和 b 赋值的两个语句,再看一个输出结果,或许对无意义的值能有所了解。

(2) 在该程序的被调用函数 other() 中,定义了一个自动类变量 a 和内部静态类变量 b,虽然变量名与主函数中的相同,但是由于它们的作用范围在各自的函数体内,相互没有影响,这是允许的。在第一次调用 other() 函数时,输出 a 和 b 的值分别为15和30。由于这里变量 b 是内部静态类的,其寿命是长的,在退出 other() 函数后仍保存其值不被释放,这便是它与自动类变量的区别。由于 b 变量的值被保存,在第二次调用 other() 函数时,变量 a 被再次定义并被初始化,它的值仍与第一次调用时一样,变量 b 不再定义和初始化,因为静态类变量仅在编译时初始化。因此, b 保留上次调用后的值,再经过 b += 20; 语句运算,其值为50。从该例中可以看到自动类变量与内部静态类变量的区别。

[例5.13] 全局变量和局部变量的应用。

分析下列程序的输出结果。

```
main()
{
    extern int i;
    int a=0;
    static int b;
    printf("i=%d,a=%d,b=%d\n",i,a,b);
    a+=5;
    other();
    printf("i=%d,a=%d,b=%d\n",i,a,b);
    i+=10;
    other();
}

int i=7;
other()
{
    static int a=2;
    int b=5;
    i+=5;
    a+=3;
    b+=1;
    printf("i=%d,a=%d,b=%d\n",i,a,b);
    b=a;
}
```

执行该程序输出结果如下:

```
i=7, a=0, b=0
i=12, a=5, b=6
i=12, a=5, b=0
i=27, a=8, b=6
```

说明:

(1) 关于外部类变量的定义和说明从该程序中可以清楚地看到由于外部类变量 i 定义在主函数的后面,在主函数中要引用变量 i 时,必须先用 extern int i; 语句进行说明。变量 i 在该

程序中的两个函数中都是可见的。因此, *i* 的值无论在主函数中被改变或者在 *other()* 函数中被改变都是有效的, 都对后面操作有影响。

(2) 关于内部静态变量的作用显然与自动类变量是不相同的。在 *other()* 函数中, *a* 是内部静态类变量, *b* 是自动类变量, 它们都是 *int* 型的。第一次调用 *other()* 函数时, 输出 *a* 值为 5, *b* 值为 6, 第二次调用 *other()* 函数时, 输出 *a* 值为 8, 但 *b* 值仍然为 6。其差别就在于 *a* 变量的寿命长, 退出 *other()* 函数后被释放。而 *b* 变量的寿命短, 退出 *other()* 函数后被释放。

[例5.14] 多文件程序中变量的作用域。

分析下列程序的输出结果。

```
int i;
void fun1(), fun2(), fun3();
main()
{
    i = 33;
    fun1();
    printf("main(): i = %d\n", i);
    fun2();
    printf("main(): i = %d\n", i);
    fun3();
    printf("main(): i = %d\n", i);
}
```

文件2(fun1.c):

```
static int i;
void fun1()
{
    i = 100;
    printf("fun1(): i(static) = %d\n", i);
}
```

文件3(fun2.c)

```
void fun2()
{
    int i = 5;
    printf("fun2(): i(auto) = %d\n", i);
    if(i)
    {
        extern int i;
        printf("fun2(): i(extern) = %d\n", i);
    }
}

extern int i;
void fun3()
{
    i = 20;
    printf("fun3(): i(extern) = %d\n", i);
    if(i)
```

```

    {
        int i=10;
        printf("fun3(),i(auto)=%d\n",i);
    }
}

```

执行该程序输出如下结果:

```

fun1(),i(static)=100
main(),i=33
fun2(),i(auto)=5
fun2(),i(extern)=33
main(),i=33
fun3(),i(extern)=20
fun3(),i(auto)=10
main(),i=20

```

说明:

(1) 该程序是由三个文件、四个函数组成的。文件 main.c 中包含了主函数 main(), 并定义了一个外部类变量 i。在该主函数中, 分别调用 fun1()、fun2() 和 fun3() 函数。文件 fun1.c 中包含了 fun1() 函数, 在该文件开头定义了一个外部静态类变量 i。文件 fun2.c 中包含了两个函数: fun2() 和 fun3()。在 fun2() 函数中, 重新定义了变量 i 为自动类变量, 在一个分程序中又说明了 i 是外部类变量。在函数 fun3() 定义前说明了 i 是外部类变量, 这里的说明可以省略, 因为该文件前面已说明过 i 是外部类变量, 并没有再重新定义。

(2) 该程序中主要是应搞清楚变量 i 在不同地方它的存储类是哪一种。

在 main() 中, i 是外部类变量。

在 fun1() 中, i 是外部静态类变量。

在 fun2() 中, i 先是自动类变量, 后是外部类变量。

在 fun3() 中, i 先是外部类变量, 后是自动类变量。

本例题输出结果请读者分析。

5.5.2 函数的存储类

函数的存储类分两种, 一种是外部类, 另一种静态类。

1. 外部函数

外部类的函数称为外部函数, 它的定义格式如下:

```

[extern]<数据类型说明><函数名>(<参数表>)
<参数说明>
{
    <函数体>
}

```

一般情况下, 说明符 extern 可以省略, 因此, 凡是定义函数时不加存储类说明符的都是外部函数。

外部函数是在程序中某个文件中定义的函数, 而在该程序的其他文件中都可以调用。

[例5.15] 外部文件的定义、说明和调用。

该程序由三个文件组成,各文件内容如下:

文件 f1.c 内容:

```
int i=1;
extern int reset(),next(),last(),new();
main()
{
    int i,j;
    i=reset();
    for(j=1;j<=3;j++)
    {
        printf("%d,%d,",i,j);
        printf("%d,",next());
        printf("%d,",last());
        printf("%d\n",new(i+j));
    }
}
```

文件 f2.c 内容:

```
static int i=10;
extern int next()
{
    return(i+=1);
}
extern int last()
{
    return(i-=1);
}
extern int new(i)
int i;
{
    static int j=5;
    return(i=j+=i);
}
```

文件 f3.c 内容:

```
extern int i;
extern int reset()
{
    return(i);
}
```

执行该程序输出如下结果:

```
1,1,11,10,7
1,2,11,10,10
1,3,11,10,14
```

说明:

(1) 该程序由三个文件、五个函数组成。

在文件 f1.c 中,首先定义了外部类变量 i。在该文件的主函数内又定义了自动类变量 i 和 j。

在文件 f2.c 中,定义了三个函数:next(),last()和 new(),在该文件开始定义了一个外部静态类变量 i,在函数 new()中,定义了一个内部静态类变量 j。

在文件 f3.c 中,文件开始说明了外部类变量 i。该文件只定义了一个函数 reset()。

(2) 该程序中有着较为丰富的存储类。搞清楚程序中变量 i 和 j 在不同函数中的存储类是十分重要的。

变量 i 在 main()函数中是自动类变量,在 next()函数中是外部静态类变量,在 last()函数中也是外部静态类变量,在 new()函数中它是一个形参,即是局部变量,在 reset()函数中是一个外部类变量。

变量 j 在 main()函数中是自动类变量,在 new()函数中它是一个内部静态类变量。

(3) 本程序中除 main()外,其余4个函数都是外部函数,在定义时没有省略 extern。

(4) 在 main()函数中,for 循环共执行4次循环体,每执行一次循环体输出一行用逗号分开的5个数。其中,i 值是不变的,j 值每次加1,next()函数的返回值为11,因为 i 是静态类的,last()函数对 i 值有影响,因此,每次调用该函数返回值总为11。同样道理,last()函数每次返回值也都为10。new()函数返回值与该函数中的局部变量 i 和 j 有关,而 j 是内部静态类的,形参 i 在每次调用时增1,j 保留上次调用的值,因此该函数返回值开始为7,第二次增3为10,第三次增4为11。

2. 内部函数

静态类的函数称为内部函数,它的定义格式如下:

static<数据类型说明><函数名>(<参数表>)

<参数说明>

```
{  
    <函数体>  
}
```

其中,static 是内部函数的说明符。该说明符不可省略。

内部函数只能在定义它的文件内调用,不能在一个文件内调用同一程序中的另一个文件中的内部函数。内部函数的作用域是文件级的,而外部函数的作用域是程序级的。

[例5.16] 内部函数的定义和调用。

分析下列程序的输出结果:

```
int i=1;  
static int reset(),next(),last(),new();  
main()  
{  
    int i,j;  
    i=reset();  
    for(j=1;j<=3;j++)  
    {
```

```

        printf("%d,%d",i,j);
        printf("%d",next(i));
        printf("%d",last(i));
        printf("%d\n",new(i+j));
    }
}
static int reset()
{
    return(i);
}
static int next(j)
int j;
{
    j=i++;
    return(j);
}
static int last(j)
int j;
{
    static int i=10;
    j=i--;
    return(j);
}
static int new(i)
int i;
{
    int j=10;
    return(i-j+=i);
}

```

说明:

- (1) 该程序由5个函数组成,一个是主函数 main(),另外4个是由主函数调用的函数:reset(),next(),last()和 new()。这4个函数被定义为内部函数,只能在该文件中调用。
- (2) 程序中变量 i 和 j 在各函数中的存储类描述如下:

	main()	reset()	next()	last()	new()
i	自动类	外部类	外部类	内部静态类	局部变量
j	自动类	无	局部变量	局部变量	自动类

- (3) 该程序的输出结果如下:

```

1,1,1,10,12
1,2,2,9,13
1,3,3,8,14

```

具体分析由读者自己完成。

练 习 题

1. C 语言中函数的定义格式是如何规定的?对函数体有何规定?

2. C 语言中函数定义后,调用之前必须要进行说明吗?如何进行说明?
3. 什么是函数的实参?什么是函数的形参?函数的形参和实参可以同名吗?
4. 什么是函数的返回值?如何实现函数的返回值?
5. C 语言中函数传递的方式是什么?传送变量值和传递变量地址值的两种调用方法有何不同?二者的特点各是什么?
6. 数组元素作函数参数和数组名作函数参数这两种调用方式有何不同?
7. 函数的嵌套调用的概念是什么?实现嵌套调用时应注意什么?
8. 什么是函数的递归调用?递归调用有何特点?
9. C 语言中标识符的作用域规则是什么?如何理解?
10. 重新定义变量的作用域规定是什么?重新定义的含义是什么?
11. 什么是存储类?变量的存储类有哪几种?函数的存储类有哪几种?
12. 自动类变量与寄存器类变量有哪些相同和不同?
13. 内部静态类变量与自动类变量有哪些相同和不同?
14. 内部静态类变量与外部静态类变量有哪些相同和不同?
15. 外部类变量与静态类变量有哪些相同和不同?
16. 外部类变量的定义和说明是一回事吗?如何定义外部类变量?如何说明外部类变量?
17. 全局变量是指哪些变量?局部变量又是指哪些变量?
18. 什么是外部函数?什么是内部函数?

作 业 题

1. 选择填空

- (1) 在由若干个文件组成的程序中,()是正确的。

A. 每个文件中只允许有一个主函数	B. 只有一个文件中有主函数
C. 所有文件中都可以没有主函数	D. 其中有一个文件中可有多个主函数
- (2) 在 C 语言程序中,()是正确的。

A. 函数的定义和调用都可以嵌套	B. 函数的定义和调用都不能嵌套
C. 函数的定义可以嵌套,而调用不能嵌套	D. 函数的定义不能嵌套,而调用可以嵌套
- (3) 函数的传值调用要求()。

A. 实参与形参类型任意,个数相等	B. 实参与形参类型完全一致,个数相等
C. 实参与形参对应类型一致,个数相等	D. 实参与形参对应类型一致,个数任意
- (4) 定义一个具有返回值的函数,但没加类型说明,该函数的默认类型为()。

A. int	B. void	C. float	D. 不确定
--------	---------	----------	--------
- (5) 函数的形参在()被分配内存单元。

A. 定义函数时	B. 编译函数时
C. 函数被调用后	D. 函数被调用前
- (6) ()的情况下,函数在调用前必须说明。

A. 定义函数时加有类型说明,并且定义在后	B. 定义函数时加有类型说明,并且定义在前
C. 具有 int 型返回值	D. 定义函数时没加任何类型说明
- (7) 函数返回值的类型取决于()。

A. 函数的类型	B. return 语句中表达式的类型
C. 调用函数将值赋给的变量的类型	D. 函数类型与 return 语句中表达式类型中较高的类

型

(8) 在 C 语言程序中,下列描述()是错误的。

- A. 被调用函数可以在所包含的文件中
- B. 被调用函数可以在同调用函数写在一个文件中
- C. 被调用函数可以在同调用函数在同一个程序的不同文件中
- D. 被调用函数可以定义在调用函数中

(9) 当一个函数调用另一个函数时,从被调用函数返回到调用函数,采用()方法是不行的。

- A. return 语句
- B. return((表达式))语句
- C. goto 语句
- D. 被调用函数的右花括号

(10) 在下列存储类说明符中,()只能用来说明存储类,而不能定义存储类变量。

- A. auto
- B. static
- C. register
- D. extern

(11) 有一个 int 型变量,在程序中使用频度很高,最好定义它为()。

- A. auto
- B. static
- C. register
- D. extern

(12) 外部静态存储类的作用域()。

- A. 与外部类一样
- B. 与内部静态类一样
- C. 在定义它的文件内并从定义时起
- D. 与自动类一样

(13) 下列关于标识符作用域的描述,()是正确的。

- A. 语句标号的作用域在定义它的函数内
- B. 自动类变量作用域在定义它的函数内
- C. 符号常量的作用域在定义它的函数内
- D. 外部静态类变量作用域在定义它的函数内

(14) 寄存器类变量的作用域与()作用域相同。

- A. 外部类变量
- B. 外部静态类变量
- C. 内部静态类变量
- D. 语句标号

(15) 下列函数 f()被调用了5次,最后 a 和 b 的值应为()。

```
f()
{
    int a;
    static int b;
    a++;
    b++;
}
```

- A. 1和5
- B. 5和5
- C. 1和1
- D. 无意义和5

2. 判断下列描述的正确性,对者划√,错者划×。

(1) C 语言中的函数包含有返回值的和无返回值的两种。

(2) 如果一个函数在定义时没加任何类型说明,函数体中又无带返回值的语句。但有的还会返回一个值,要使该函数不返回值,需加 void 说明符。

(3) 在对函数采用原型说明时,如果实参类型与形参类型不一致时,会产生报错信息。

(4) 函数的说明只能在某个函数体中,不能放在函数体外。

(5) 无返回值的函数,在调用之前一定不要说明。

(6) 函数的实参可以是表达式,形参也可以是表达式。

(7) 在 C 语言程序中,除了主函数 main()外,任何其他函数都可以被调用。

(8) 函数的函数体不能为空,至少要有有一个空语句。

(9) 递归调用的特点是提高程序的执行速度。

(10) 在函数调用中,如果实参用地址值,形参一定要用指针,这种调用可在被调用函数中通过改变形参所指向的内容在改变实参变量值。

(11) 在一个程序中,某个外部变量可以说明多次,但只能定义一次。

(12) 自动类变量寿命短是指它退出作用域外,该变量被释放。

(13) 内部静态类变量的作用域与外部类的相同,因此,它们的寿命都是长的。

(14) 外部类变量作用域大,寿命又长,因此在编程中应该尽量多的使用外部类变量。

(15) 定义外部静态类变量只能写在文件开头,不能写在中间。

(16) 在某个函数体开头定义了一个自动类变量,那么它在该函数体中任何部分(包括内部嵌套的分程序内)都可以引用它,即改变它的值。

(17) 函数的形参是属于局部变量的,它的作用域是它所被定义的函数体内,因此两个不同的函数可以使用相同的形参。

(18) 如果一个变量在它的作用域内可见并且存在,而出了该作用域后,则不可见并且也不存在,那么这种变量一定是自动类的。

(19) 只有存储类为静态的和外部的数组才能被赋初值,并且在编译时执行。

3. 分析下列程序的输出结果。

(1)

```
void f(n)
int n;
{
    int x=5;
    static int y=10;
    if(n>0)
    {
        ++x;
        ++y;
        printf("%d,%d\n",x,y);
    }
}
main()
{
    int m=1;
    f(m);
}
```

(2)

```
main()
{
    int i;
    for(i=1;i<=4;i++)
        f(i);
}
f(j)
int j;
{
    static int a=10;
    int b=1;
    b++;
    printf("%d+%d+%d=%d\n",a,b,j,a+b+j);
    a+=10;
}
```

(3)

```
main()
{
```

```

    extern int x,y;
    printf("%d\n",add(x,y));
}
int x=20,y=-5;
add(a,b)
int a,b;
{
    int s;
    s=a+b;
    return(s);
}

```

(4)

```

main()
{
    int i,s;
    s=0;
    for(i=1;i<=5;i++)
        s+=fac(i);
    printf("5!+4!+3!+2!+1!=%d\n",s);
}
fac(a)
int a;
{
    static int b=1;
    b*=a;
    return(b);
}

```

(5)

```

#define N 5
void fun();
main()
{
    int i;
    for(i=2;i<N;i++)
        fun();
}
void fun()
{
    static int a;
    int b=2;
    printf("%d\n", (a+=3,a+b));
}

```

(6)

```

main()
{

```

```

    int x,y,z;
    fun(5,6,&x);
    fun(7,x,&y);
    fun(x,y,&z);
    printf("%d,%d,%d\n",x,y,z);
}
fun(a,b,c)
int a,b,*c;
{
    b+=a;
    *c=b-a;
}

```

(7)

```

int a[]={1,3,5,7};
main()
{
    int i,y=2;
    for(i=0;i<4;i++)
    {
        y=fun(a,y);
        printf("%d",y);
    }
    printf("\n");
}
fun(b,y)
int b[],y;
{
    static int a;
    y=b[a]+1;
    a++;
    return(y);
}

```

(8)

```

main()
{
    int a=5,b=10;
    printf("a=%d,b=%d\n",a,b);
    swap1(a,b);
    printf("a=%d,b=%d\n",a,b);
    swap2(&a,&b);
    printf("a=%d,b=%d\n",a,b);
}
swap1(x,y)
int x,y;
{
    int temp;

```

```

    temp=x;
    x=y;
    y=temp;
    printf("x=%d,y=%d\n",x,y);
}
swap2(x,y)
int *x,*y;
{
    int z,*temp;
    temp=&z;
    *temp=*x;
    *x=*y;
    *y=*temp;
    printf("x=%d,y=%d\n",*x,*y);
}

```

(9)

```

main()
{
    int a,b;
    a=5;
    b=8;
    printf("sum1=%d\n",add(a,b));
    printf("sum2=%d\n",add(a,add(a,b)));
    printf("sum3=%d\n",add(a,add(b,add(a,b))));
}
add(x,y)
int x,y;
{
    return(x+y);
}

```

(10)

```

#define N 6
main()
{
    int a=N;
    printf("%d\n",f(a));
}
f(a)
int a;
{
    return((a==0)?1:a*f(a-1));
}

```

4. 按下列要求编程,并上机验证。

- (1) 从键盘上输入10个浮点数,求出其和及平均值。要求写出求和及求平均值的函数。
- (2) 从键盘上输入若干个整数,去掉重复的,将剩余的数由大到小排序输出。

- (3) 给定某个年、月、日的值,计算出属于该年的第几天?要求写出计算闰年的函数和计算日期的函数。
- (4) 写出一个函数,使从键盘上输入的一个字符串反序存放,并在主函数中输入和输出该字符串。
- (5) 编写两个函数:一个是将从键盘上输入的5位整数转换成每个数字之间加一个空格符的字符串;另一个是求出转换的字符串长度。由主函数输入和输出。
- (6) 输入5个学生4门功课的成绩,然后求出:① 每个学生的总分;② 每门课程的平均分;③ 找出总分最低的学生,要求输出学生姓名和总分数。
- (7) 写一个函数,将十六进制数转换成十进制数。
- (8) 使用递归调用方法将一个 n 位整数转换成字符串。

第六章 预处理功能和类型定义

本章主要讲解 C 语言所提供的预处理功能和类型定义两个问题。这是 C 语言为了使编程简洁、清晰和使程序含意明了所提供的两种功能。预处理功能包含有宏定义、文件包含和条件编译三个主要部分。类型定义是通过使用类型定义语句将已有类型定义成新的标识符所代表的类型。这两种功能编程者将根据需要进行选用。

6.1 预处理功能概述

本节主要讲述预处理功能的特点。

预处理功能是由很多预处理命令组成,这些命令将在编译时进行通常的编译功能(包含词法和语法分析、代码生成、优化等)之前进行处理,故称为“预处理”。预处理后的结果和源程序一起再进行通常的编译操作,进而得到目标代码。

预处理功能主要包括如下三种:宏定义、文件包含和条件编译。

这些功能是通过相应的宏定义命令、文件包含命令和条件编译命令来实现。这些命令不同于 C 语言的语句,因为它们具有如下的特点:

- (1) 多数预处理命令只是一种替代的功能,这种替代是简单的替换,而不进行语法检查。
- (2) 预处理命令都是在通常的编译之前进行的,编译时已经执行完了预处理命令,即对预处理后的结果进行编译,这时进行词法和语法分析等通常的程序编译。
- (3) 预处理命令后面不加分号,这也是在形式上与语句的区别。
- (4) 为了使预处理命令与一般 C 语言语句相区别,凡是预处理命令都以井号("#")开头。
- (5) 多数预处理命令根据它的功能而被放在文件开头为宜,但是根据需要,也可以放到文件的其他位置。不要产生错觉,好像所有的预处理命令都必须放在文件开头。

学习和掌握预处理功能时,应该了解它的上述规定,以便正确地使用和理解这些预处理命令。

6.2 宏定义

宏定义是最常用的预处理功能之一。宏定义分为两种,一种是简单的宏定义,它不带有参数,另一种是带参数的宏定义。无论是哪种宏定义,它都是将一个标识符或称宏名定义为一个字符串或称宏体。在程序中出现的是标识符(或宏名),在编译之前进行宏替换,即将标识符(或宏名)被替换成为所定义的字符串(或宏体)。下面分别介绍简单宏定义和带参数宏定义的特点和功能。

6.2.1 简单宏定义

1. 简单宏定义的格式和功能

简单宏定义的格式如下:

```
#define <标识符> <字符串>
```

其中,define 是关键字,它表示该命令为宏定义,<标识符>是宏名,它的写法同标识符。
<字符串>用来表示<标识符>所代表的字符串。

简单宏定义是定义一个标识符(宏名)来代表一个字符串。

前面讲过的符号常量就是用这种简单宏定义来实现的。例如:

```
#define PI 3.14159265
```

这是一条宏定义的命令,它的作用是用指定的标识符 PI 来代替字符串“3.14159265”。在程序中出现的是 PI,在编译前预处理时,将所有的 PI 都用“3.14159265”来代替,即使用宏名来代替指定的字符串。这一过程又称为“宏替换”或称为“宏展开”。

[例6.1] 给出半径求圆的面积。

```
#define PI 3.14159265
main()
{
    float r, s;
    printf("Input radius: ");
    scanf("%f", &r);
    s=PI*r*r;
    printf("s=%.4f\n", s);
}
```

执行该程序,出现如下信息:

Input radius: 5

输出结果如下:

s=78.5398

说明:该例中,开始定义了符号常量 PI,它是用宏定义来实现的。程序中出现的 PI,在编译前预处理时将用“3.14159265”来替换。

2. 使用简单宏定义时的注意事项

(1) 宏定义中的<标识符>(即宏名)一般习惯用大写字母,以便与变量名区别。这样,在 C 语言程序的各表达式语句中,凡是大写字母的标识符(指全部大写字母)一般是符号常量。但是,宏定义中的宏名也可以用小写字母。

(2) 宏定义是预处理功能中的一种命令,它不是语句,因此,行末不需加分号。如果加了分号,则该分号将作为所定义的字符串的一部分,即按字符串的一部分来处理。

(3) 宏替换是一种简单的代替,替换时不作语法检查。如果所定义的字符串中有错,例如,将数字 0,误写为字母 o,预处理照样代换,并不报错,而在编译中进行语法检查时才报错。因此,要记住宏替换操作只是简单的代换,用宏定义时的字符串来替换其宏名。

(4) 宏定义中宏名的作用域为定义该命令的文件中,并从定义时起,到终止宏定义命令(#undef<标识符>)为止,如果没有终止宏定义命令,则到该文件结束为止。通常放在文件开

头,表示在此文件内有效。

终止宏定义命令的格式如下:

`#undef<标识符>`

其中,undef 是关键字,<标识符>表示要终止的宏名,该宏名是在该文件中已定义的标识符。例如:

`#undef PI`

表示宏定义的 PI 到此终止,即终止后的 PI 不再代表所定义的字符串了。

(5) 宏定义可以嵌套。所谓嵌套是指在进行宏定义时,可以引用已定义的宏名。例如:

`#define WIDTH 10`

`#define LENGTH (WIDTH+10)`

`#define AREA (LENGTH * WIDTH)`

在第二个宏定义中引用了第一个宏定义的宏名 WIDTH,而在第三个宏定义中引用了第一个宏定义的宏名 WIDTH 和第二个宏定义的宏名 LENGTH。第二个和第三个宏定义便是宏定义的嵌套使用。嵌套的宏定义在替换时,要进行层层替换。例如,在上述宏定义的文件中,出现如下语句:

`s=AREA+50;`

则替换步骤如下:

先替换 AREA,结果如下:

`s=(LENGTH * WIDTH)+50;`

再替换 LENGTH,结果如下:

`s=((WIDTH+10) * WIDTH)+50;`

最后替换 WIDTH,结果如下:

`s=((10+10) * 10)+50;`

(6) 一般编译系统对于加有双引号的字符串的宏名不予替换。但是,有的编译系统对字符串的宏名也予替换。使用时应该注意该编译系统对字符串内宏名的处理规则。

(7) 一般编译系统在宏替换时,隐含一空格符,即用空格符将前后两部分分开。有的编译系将不隐含空格符。

下面举几个例子,对宏定义的使用作进一步说明。

[例6.2] 分析下列程序输出结果,并说明简单宏定义在本程序中的应用。

```
#define A -a
#define TWOA 2 * A
main()
{
    int a=1;
    printf("TWOA = %d\n", TWOA);
    printf(" %d\n", -A);
}
```

执行该程序输出结果如下:

TWOA = -2

1

说明:

(1) 该程序开头有两个宏定义命令,并且使用宏定义嵌套的方法。

(2) 在 Turbo C 编译系统中,对加双引号的字符串内的宏名不予替换。即 printf()函数中,控制串“TWOA=%d\n”中的 TWOA 不被替换,而参数 TWOA 被替换为

2 * A

进一步替换为

2 * -a

这里,a 为1,上述结果为-2。

又在第二个 printf()函数中,其参数-A 被替换为

- -a

其值为1。这里可以看出 Turbo C 编译系统在宏替换时,加有隐含空格,其值才为1。否则,替换后为

- -a

其值为0。有的编译系统确有此种结果。

[例6.3] 分析下列程序的输出结果,并说明宏名的作用域。

```
main()
{
    #define N 5
    printf("N=%d\n",N);
    #define M N+3
    printf("M=%d\n",M);
    #undef N
    #define N 10
    printf("new M=%d\n",M);
}
```

执行该程序输出结果如下:

N=5

M=8

new M=13

说明:

(1) 宏定义命令不一定必须写在文件开头,可以根据需要写在文件的任何位置。该程序中在不同位置出现了3条宏定义命令和1条终止宏定义命令。

(2) 宏名的作用域是在定义它的文件内,并从定义时开始,到终止定义时为止。本例中,宏名 N 从定义时开始起作用,到 #undef N 命令为止。本例中,又再定义 N 到文件结束。可见,对一个宏名进行重新定义之前必须先将原定义取消。而本例中宏名 M 从定义时起作用,直到文件结束。

6.2.2 带参数的宏定义

1. 带参数宏定义的格式和功能

带参数宏定义的一般格式如下:

`#define <宏名>(<参数表>)<宏体>`

其中,<宏名>是标识符,一般习惯上用大写字母,<参数表>是由一个或多个参数组成的,多个参数之间用逗号分隔,<宏体>是一个字符串,其中包含<参数表>中所指定的参数,它可以是由若干个语句组成的。该命令末尾一般不加分号。

带参数的宏定义在宏替换时,不是简单的用宏体替换宏名,而是用“实参”替换“形参”。这里所说的实参是指程序中引用宏名的参数,而形参是指宏定义时,宏名后边参数表中的参数。例如,

```
#define SQ(x) x * x
```

在程序中出现下述语句:

```
a = SQ(5);
```

这里,在宏定义中参数 x 是形参,而程序中 $SQ(5)$ 的 5 是实参,宏替换时,将用 5 来替换 x ,其结果如下:

```
a = 5 * 5;
```

又例如,

```
#define ADD(x, y) x + y
```

在程序中出现下述语句:

```
a = ADD(5, 3);
```

宏替换后结果如下:

```
a = 5 + 3;
```

如果在上述宏定义下,程序中出现如下语句:

```
b = ADD(a + 1, b - 1);
```

宏替换后结果如下:

```
b = a + 1 + b - 1;
```

由此可见,带参数的宏定义是这样替换的:按照宏定义中所指定的宏体从左至右用程序中出现的宏的实参来替换宏体中的形参,对非参数字符,则保留。宏中的实参可以是常量、变量或表达式。

2. 使用带参数的宏定义应该注意的事项

(1) 在宏定义时,宏名与左圆括号之间不能出现空格符,否则空格符后将作为宏体的一部分。例如:

```
#define ADD (x, y) x + y
```

将认为 ADD 是不带参数的宏名,而字符串 $(x, y) x + y$ 作为宏体。显然,这不是原来的含意。因此,宏名后与左圆括号间一定不能加空格符。

(2) 宏体中,各参数上加括号是十分重要的。例如:

```
#define SQ(x) x * x
```

当程序中出现下列语句,

```
a = SQ(a + 1);
```

替换后,则为:

```
a = a + 1 * a + 1;
```

而不能将替换的结果写成

```
a=(a+1)*(a+1);
```

如果要将替换后结果写成上述形式,则需要将宏定义改写为:

```
#define SQ(x) (x)*(x)
```

由此可见,在宏定义中,对宏体内的形参外向加上括号是很重要的,它可以避免在优先级上可能出现的问题。

对上述宏定义最好写成下述形式:

```
#define SQ(x) ((x)*(x))
```

这里的圆括号是很有用的。例如,在有如下语句时,

```
m=50/SQ(b+1);
```

替换后结果如下:

```
m=50/((b+1)*(b+1));
```

(3) 带参数的宏定义与函数的区别

带参数的宏定义和函数尽管在形式上非常相似,特别是当宏名使用小写字母时,出现在程序中很难区别出来是带参数的宏定义还是函数。但是,这二者是根本不同,它们之间的区别概述如下:

① 定义形式上不同。带参数的宏定义的定义格式前面讲过了,它是通过预处理命令 `define` 来定义的。它的作用域是在定义它的文件内,并从定义处开始。而函数的定义格式在本书“函数和存储类”一章中描述过了,它的作用域分为程序级的和文件级的两种。

② 处理时间上不同。宏定义是在编译预处理时处理的,处理后再进行编译。而函数是在执行时处理的。它们二者占用的是不同阶段的时间。

③ 处理方式上不同。带参数的宏定义在进行宏替换时,用实参来代替形参,这里只是简单替换,并不做语法上的检查。而函数调用时,是将实参的值赋给形参,要求对应类型一致。宏定义中在参数替换时,不要求类型一致。

④ 时间和空间的开销上不同。带参数的宏定义,是在通常的编译之前完成替换的,因此它在该程序的目标代码的形成上并没有影响。函数调用是在执行时进行的,因此,采用函数调用的方式可以减少该程序的目标代码,所以,在空间的开销上可以减少。但是,函数调用要有额外的时间上的开销。因为调用前要保留现场,调用后又要恢复现场,因此函数调用时间开销要比带参数的宏定义大。带参数宏定义在使用中比函数时间开销小是一个重要特征。

⑤ 类型的要求上不同。带参数的宏定义对形参的类型不必说明,它没有类型的限制。而函数的形参在定义时必须进行类型说明。例如,两个数相减的操作用带参数的宏定义和函数分别定义如下:

用带参数的宏定义格式如下:

```
#define MINU(x,y) (x)-(y)
```

用函数的定义格式如下:

```
minu(x,y)
```

```
int x,y;
```

```
{
```

```
    return(x-y);
```

```
}
```

该函数只能进行两个 int 型数值相减的运算。而上述的宏定义可以进行两个 char 型量的相减,也可以进行两个 int 型数的相减,还可以进行两个 float 型数的相减,因为它没有类型的限制。

通过上述对于带参数的宏定义和函数之间区别的分析,不难看出两者各有特点。对于同一个问题可以采用两种不同的表示形式,那么到底选择哪一种更好些呢?一般说来,在功能比较简单的情况下,选用带参数的宏定义能更好些,特别是在需要反复引用的情况下,用宏定义的时间开销较少。例如,比较两个整数的大小,并输出最大的,可用如下的宏定义来实现:

```
#define fl(a, b) printf("%d\n", a>b?a: b);
```

又例如,计算一个自然数的立方值,可用如下的宏定义实现:

```
#define f2(a) printf("%d\n", a * a * a);
```

功能比较复杂的还是选用函数来定义。

6.2.3 宏定义的应用

在 C 语言程序中,宏定义主要用于下述几个方面。

(1) 符号常量的定义

C 语言中的常量一般都用符号常量表示,这样不仅书写简便,而且易于修改、易于移植,还可以使标识符有更明显的含意。例如,

```
#define PI 3.14159265
```

```
#define E 2.71828
```

```
#define EPS 1.0e-9
```

```
#define MAX 32767
```

```
#define TRUE 1
```

```
#define FALSE 0
```

等等。

(2) 功能简单使用频繁的情况下,用带参数的宏定义比用函数可以提高速度。因为函数调用要花费额外的时间开销。

(3) 在多数情况下,为了书写简练,程序易读而采用宏定义。

[例6.4] 分析下列程序输出结果,并说明宏定义在该程序中所起的作用。

```
#define F1 "%d\n"
#define F2 "%d,%d\n"
#define PR1(a) printf(F1,a)
#define PR2(a,b) printf(F2,a,b)
main()
{
    int x,y,z;
    x=5;
    y=5*x;
    z=x+y;
    PR1(x);
    PR2(y,z);
    PR1(2*y);
}
```

```
    PR2(x+5,z/5);  
}
```

执行该程序输出结果如下：

```
5  
25,30  
50  
10,6
```

说明：

(1) 该程序前面有4条宏定义命令。前两条是简单宏定义，后两条是带参数的宏定义。这4条宏定义命令都是有关标准文件的格式输出的。

(2) 程序中的最后4条语句是输出显示结果的语句，由于引用了宏定义命令，使得该输出语句简练、清晰。由于在C语言程序中，输出显示结果语句用得较多，并且可选用不同的参数个数，如果能使用宏定义命令使之简化，将给书写带来方便。特别是这里使用一次文件包含命令（下面讲述）将宏定义部分写在一个.h（头文件）文件内，程序需要时只要包含进去就可以了，这样将会更加简便。

6.3 文件包含

文件包含的含意是在一个文件中可以包含另外一个文件的内容。这时，这个文件将由两个文件组成。文件包含命令将会实现这一功能。

6.3.1 文件包含命令的格式和功能

文件包含命令是C语言程序常用的一条预处理命令，它的格式如下：

```
#include(文件名)
```

其中，include 是关键字，(文件名)是被包含的文件名，这里要求使用文件全名，包括路径名和扩展名。

文件包含的功能就是将指定的被包含文件的内容放置在文件包含命令出现的地方。该命令可写在程序的任一位置，但是一般写在一个文件的开头，这是一种明智的选择。因为该命令在何处出现，所包含的文件内容就被放置到该命令出现的位置，这就相当于将被包含文件的内容插入到该命令的地方。而被包含的.h文件中往往是一些该程序所需要的一些说明或定义，它包括符号常量的定义、类型定义、带参数的宏定义、数组、结构、联合和枚举的定义等等，它还可以包含外部变量的定义、函数的定义和说明等等。这些内容程序中将要用到，显然放在程序前面比放在中间和后面要好些。

文件包含的用途在于减少程序人员的重复劳动，使得C语言程序更加简洁，提高程序的可读性。但是，如果对文件包含命令使用不当，会增加程序的代码长度，包含了一些该程序所不需要的内容。因此，选择包含的文件时要慎重，定义被包含的文件时要短小。

[6.5] 将例6.4中的宏定义部分写在一个.h文件中，当程序需要时，将它包含进去。其做法如下：

定义.h文件名为 print.h，它将包含有关输出显示结果的宏定义，其内容如下：

```
#define F1 "%d\n"
#define F2 "%d,%d\n"
#define PR1(a) printf(F1,a)
#define PR2(a,b) printf(F2,a,b)
```

程序内容如下：

```
#include "print.h"
main()
{
    int x,y,z;
    x=5;
    y=5*x;
    z=x+y;
    PR1(x);
    PR2(y,z);
    PR1(2*y);
    PR2(x+5,z/5);
}
```

执行该程序的输出结果与例6.4的结果相同。

该程序中在开头使用了文件包含命令,将文件 print.h 包含在该程序的开头。从程序书写角度来看只使用了一条文件包含命令便可代替该文件(指 print.h 文件)中的4条宏定义命令,可见书写简单了。print.h 文件还可以用于其他程序中,如果某个程序需要该文件的内容,只要使用一条文件包含命令就可以了。

6.3.2 使用文件包含命令时应注意事项

(1) 一个文件包含命令只能包含一个文件。如果某个程序需要包含多个文件,则可使用多条文件包含命令。例如,下列写法是错误的,

```
#includ "xy.h" "mn.h"
```

而改写成下列形式是正确的,

```
#include "xy.h"
#include "mn.h"
```

其前后顺序取决于被包含文件内容间的相互关系。如果两个被包含文件内容相互无关,则书写的前后顺序也无关。如果一个文件中要使用另一个文件的内容,则另一个文件要写在前面。

(2) 文件包含的定义是可以嵌套的。所谓文件包含定义的嵌套是指在一个被包含的文件中还可以包含有其他文件。例如,

文件 file1.c 的内容如下:

```
#include "file2.h"
```

```
⋮
```

而被包含的文件 file2.h 的内容如下:

```
#include "file3.h"
```

```
⋮
```


这便是文件包含的嵌套,它可以等价于:

文件 file1.c 的内容如下:

```
#include "file3.h"
#include "file2.h"
    :
```

按上述顺序写比较安全。

(3) 文件包含命令中,引用的被包含文件有下面两种方式:

方式一:

```
#include <文件名>
```

方式二:

```
#include "文件名"
```

这两种方式中,文件名都要求是全名,方式一中用一对尖括号将文件名括起,它表示所引用的被包含文件是系统提供的,并放在指定目录下的.h 文件。例如,

```
#include <stdio.h>
```

```
#include <math.h>
```

等等。

这种方式引用时,系统将到指定的目录去查找系统提供的.h 文件。

方式二是用一对双引号将文件名括起,它表示所引用的被包含文件可能是用户自己定义的文件,并放在当前目录下或系统可自动查找的目录下。对这种引用的文件,系统先在当前目录下或用 DOS 命令(PATH)连起来的通用目录下查找,如果找不到再到系统指定的目录下查找。因此,要求编程人员对自己定义的被包含文件,引用时使用双引号,而系统提供的.h 文件最好使用尖括号,这样查找可能快些。

(4) 被包含的文件可以是任意的源文件,不一定必须是.h 文件,也可以是.c 文件。

[例6.6] 使用文件包含命令将一个.c 文件包含在另一个.c 文件之中。

f1.c 文件内容如下:

```
add(x,y)
int x,y;
{
    int z;
    z=x+y;
    return(z);
}
```

f2.c 文件内容如下:

```
#include "f1.c"
main()
{
    int a,b;
    a=5;
    b=10;
    printf("%d\n",add(a,b));
}
```

执行 f2.c 文件后,将获得如下结果:

15

在 f2.c 文件的开头就使用文件包含命令将 f1.c 文件包含了进去,这就相当于将 f1.c 文件的内容插入到 f2.c 文件中 main()的前边。

(5) 为提高程序的可移植性,编程时事先将一些在不同环境或条件下需要修改的量或需要修改的部分放在一个被包含的文件中,以后为适应某种环境需要修改时,可以只修改包含文件中的内容,这将为该程序的移植提供了方便。

6.4 条件编译

条件编译指的是对编译的源程序的某种控制。在某种条件下,源程序中的这些行参加编译;而在另一种条件下,源程序中那些行参加编译。即根据不同的条件来控制源程序参加编译的内容,因此称为条件编译。

6.4.1 条件编译的常用命令格式

条件编译的常用命令格式有如下三种。

格式一:

```
#ifdef<标识符>
    <程序段1>
#else
    <程序段2>
#endif
```

其中,ifdef, else 和 endif 是关键字。<标识符>是指是否使用 #defin 命令定义过。<程序段1>和<程序段2>是由语句或命令组成的程序段。

该格式的功能描述如下:

当<标识符>被宏定义时,<程序段1>中的语句或命令参加编译;否则<程序段2>中的语句或命令参加编译。该格式中,#else 是可以省略的,则变成下列格式:

```
#ifdef<标识符>
    <程序段>
#endif
```

格式二:

```
#ifndef<标识符>
    <程序段1>
#else
    <程序段2>
#endif
```

该格式与格式一大致相同,只是第一个关键字中不一样。

该格式的功能是:如果<标识符>未被宏定义时,则编译<程序段1>,否则编译<程序段2>。

格式三:

```
#if<表达式>
    <程序段1>
#else
    <程序段2>
#endif
```

该格式中,第一行与前面二种格式不同,if 是关键字,<表达式>是常量表达式。

该格式的功能是:当<表达式>是有非0值时,则<程序段1>参加编译;否则<程序段2>参加编译。当#else 省略了,则变成如下简化格式:

```
#if<表达式>
    <程序段>
#endif
```

该格式中决定参加编译的程序段取决于<表达式>的值是零还是非零。

以上三种格式可根据需要进行选择。

6.4.2 条件编译命令的应用

(1) 条件编译可以用来提高程序的通用性。由于不同的计算机存在一定差异,有的是16位机,有的是32位机,为使一个源程序在不同机器上运行,可以使用以下的条件编译(假定区别仅在机器的字长上):

```
    :
    #ifdef PC16
    #define INTSIZE 16
    #else
    #define INTSIZE 32
    #endif
    :
```

如果 PC16在前面已被宏定义过,例如以下命令:

```
#define PC16 16
```

则编译下面的命令行:

```
#define INTSIZE 16
```

否则,PC16在前面没有被宏定义过,则编译下面的命令行:

```
#define INTSIZE 32
```

这里,到底编译哪些行,关键在于标识符 PC16是否被宏定义,而定义为多少并没有关系。

前面介绍的只是一种简单情况,其思路是将其不同用语句或命令表示出来,根据不同的要求进行选择,按照这种思路来设计条件编译。

(2) 条件编译会给程序调试带来方便。在源程序的调试中,常常需要跟踪程序的执行情况,为此,在程序中加一些输出信息的语句,通过这些输出信息来跟踪判断程序是否有错误。这是一种常用的调试手段。在调试结束后,需要把为了调试而增加的那些输出信息的语句删除掉,然而这种删除工作常常带来一些不方便,因为不能多删,也不能少删,必须删对,否则又

将出错。如果使用条件编译将会带来一些方便。即在满足调试条件的情况下,加上一些输出信息的语句行,在调试完成后,只要改变其条件使之不满足调试条件,这时再重新编译,原来加进去的一些输出信息的语句将不再被编译,这相当于被删除了,这种“自动”删除要方便得多。例如,在调试某个程序时,为得出输出信息以供判断,可在源程序中插入以下条件编译:

```
    :
    #if DEBUG
        printf("a=%d, b=%d\n", a, b);
    #endif
    :
```

在调试程序时,在它们前面设置 `#define DEBUG 1`,则 `printf()` 语句行将参加编译,于是将输出供判断参考的 `a` 和 `b` 的值。当调试完成后,只是将前面的设置修改为: `#define DEBUG 0`,则 `printf()` 语句行将不参加编译,就相当于该行被删除一样。可见,使用条件编译通过改变其条件来“删除”不需要参加编译的程序段是很方便的。

(3) 使用条件编译替代 `if` 语句会减少目标代码的长度。

[例6.7] 将一个已知的十进制数,根据设置条件编译的不同条件,使它按十六进制输出,或按八进制输出。

```
main()
{
    #define N 1;
    int n=55;
    #ifdef N
        printf("%x\n",n);
    #else
        printf("%o\n",n);
    #endif
}
```

由于程序中, `N` 已被宏定义,因此执行该程序输出结果如下:

37

这是55的十六进制表示的值。

如果删除程序中的 `#define N 1` 这行后,执行该程序输出结果则为55的八进制表示值:

67

前面已经讲述了三种常用的预处理命令:宏定义命令、文件包含命令和条件编译命令。此外,还有一些其他不常用的预处理命令本书就不再讲述了。下面以一个使用上述三种预处理命令的程序来结束本节的内容。

[6.8] 分析下列程序的输出结果,并分析各种预处理命令的用法。

该程序内容如下:

```
#define N 1
#define ABC main(){printf("Hello! %s\n",s);}
#include "abc.h"
```

其中, `abc.h` 文件的内容如下:

```
#if N
    char s[]="good morning!";
    ABC
#endif
```

执行该程序后输出结果为：

Hello! good morning!

说明：

(1) 在该程序中,有两条宏定义命令,都是简单宏定义,有一条文件包含命令,所包含的文件是用户定义的。

(2) 在被包含的文件 abc.h 中,只有一个条件编译命令,该命令表明,只要是 N 为非0,则给出的程序段将被编译。如果 N 为0,则无编译的程序段。

(3) 在该程序编译前,进行预处理时,先将被包含文件插入到包含文件命令处,再经过进行条件编译命令和宏定义命令后,该程序将被替换成如下待编译的格式：

```
char s[]="good morning!";
main()
{
    printf("Hello!%s\n", s);
}
```

该程序经过编译,执行后将获得上述结果,这是很自然的。

6.5 类型定义

6.5.1 类型定义的含意和类型定义语句

1. 类型定义的含意

什么是类型定义?是再定义新的类型吗?不是。这里的类型定义指根据某种需要对已有的类型用一种新的类型名字来替代,即给已有的或者已存在的类型起个新名。而不是再重新定义原来没有的新类型。

2. 类型定义语句

类型定义是通过类型定义语句来实现的。类型定义语句格式如下：

```
typedef <已有类型说明符> <新类型名表>;
```

其中,typedef 是关键字。<已有类型说明符>包含 C 语言中规定的所有合法的类型说明符和已用类型定义语句定义的新的类型名,<新类型名表>是用类型定义语句定义的新的类型说明符,该表中可以有一个或多个新的类型说明符,多个新的类型说明符用逗号分隔。

该语句在使用时应该注意如下事项：

- (1) 该语句只是对已有的类型定义一个新名,而绝不是产生一种新的类型。
- (2) 使用该语句时,习惯上将新定义的类型名用大写字母,以便与系统原有的类型名加以区别。这里如用小写字母也可以。
- (3) 类型定义可以嵌套。所谓嵌套是指用该语句定义的新的类型名可再用来定义新的类型。例如,

```
typedef int FEET;
typedef FEET *POINT;
POINT p1, p2;
```

这里,FEET 是用 typedef 语句定义出的新类型名,接着又用 FEET 来定义新类型 POINT,由于 FEET 为 int 型的类型,而 POINT 为指向整型类型的指针,然后,又用 POINT 定义两个变量 p1 和 p2,这时,p1 和 p2 都是指向 int 型类型的指针。

(4) 常把使用该语句定义的数据类型包含在某个头文件中,这样在不同的源文件中用到该类型时,可使用文件包含命令将所需的头文件包含到该源文件中。

(5) 在使用中会看到 typedef 语句与 #define 有相似之处,但是二者实际上是很不相同的。其中, #define 在编译的预处理时完成,只是简单的替换,而 typedef 语句在编译时处理,它并不是替换。下面通过一个例子加以说明。

[例6.9] 观察下列两个程序的不同,并分析这两个程序输出结果。上机运行后将会发现什么问题,从中理解 typedef 语句与 #define 命令的不同。

程序1内容如下:

```
typedef int INT;
main()
{
    INT a,b;
    a=5; b=6;
    printf("a=%d,b=%d\n",a,b);
    {
        float INT;
        INT=3.0;
        printf("2*INT=%f,2f\n",2*INT);
    }
}
```

程序2内容如下:

```
#define INT int
main()
{ INT a,b;
  a=5; b=6;
  printf("a=%d,b=%d\n",a,b);
  {
    float INT;
    INT=3.0;
    printf("2*INT=%f,2f\n",2*INT);
  }
}
```

说明:

- (1) 从这两个程序的内容上看除了第一行外,其余内容是相同的。
- (2) 运行这两个程序时,程序1的输出结果如下:

```
a=5, b=6
2 * INT=6.00
```

而运行程序2时,编译时报错。指出程序中的 float INT 语句在说明中有太多的类型。该程序只有做一下修改,即在 printf() 语句后面加一条取消 INT 宏定义的语句,才可通过编译,并得到与程序1相同的结果。所加语句的格式如下:

```
#undef INT
```

从这里可以看到,typedef 语句与 #define 命令是不相同的。

6.5.2 类型定义的应用

(1) 使用类型定义可以简化书写,将一个复杂的类型定义成一个简单的新类型名。例如,

```
typedef struct date {
    int month, day, year;
}DATE ;
```

用一个新的类型名 DATE 来代表一个复杂的结构类型。再用 DATE 来定义结构变量,例如:

```
DATE day, *p;
```

其中,day 是一个具有 date 结构类型的结构变量,p 是一个指向具有 date 结构类型的结构变量的指针。这样,使得书写更加简洁。

(2) 使用类型定义会给一些变量带来更多的有用信息。正如大家所知道的,在下列的说明中,

```
int a, b, c;
```

只能知道 a, b 和 c 是三个整型变量。如果使用如下的类型定义:

```
typedef int FEET, INCHES;
```

使用新的类型名 FEET 和 INCHES 来定义变量,

```
FEET a, b;
```

```
IECHES c;
```

这时,不仅知道 a, b 和 c 是 int 型变量,而且还可知 a 和 b 这两个变量是用来存放以英尺为单位的长度值的,而 c 是用来存放以英寸为单位的长度值的,因此,使用类型定义是给某些变量增加一些有用信息的。

(3) 使用类型定义,用新类型定义的变量在编译时做类型检查,从而增加安全性。

练 习 题

1. C 语言中常用的预处理命令有哪些种?预处理命令有何特点?
2. 什么叫宏定义?什么叫宏替换?
3. 简单宏定义的格式如何?使用简单宏定义时应注意哪些事项?
4. 带参数的宏定义的格式如何?使用带参数的宏定义时应注意哪些事项?
5. 带参数的宏定义与函数有哪些差别?为实现某一功能是选择带参数的宏定义的形式还是选择函数的形式的依据是什么?

6. 带参数宏定义中的形参与引用宏定义时的实参有何关系?
7. 文件包含命令的格式如何?被包含文件的两种引用方式有何区别?
8. 文件包含的含意是什么?它有何作用?
9. 条件编译的含意是什么?条件编译命令常用的几种格式如何?
10. 类型定义的含意是什么?类型定义又有何用处?

作业题

1. 选择填空:

- (1) 预处理命令在程序中都是以()开头。
A. # B. * C. > D. <
- (2) 在使用 include 命令时,被包含的文件()作扩展名。
A. 必须以.h B. 不能用.h C. 不一定以.h D. 必须以.c
- (3) 下面定义以 AB 为标识符的符号常量,()是正确的。
A. #define 7.21 AB
B. define AB 7.21
C. #define AB 7.21
D. #include AB 7.21
- (4) 带参数的宏定义中,形参的个数()。
A. 只能有1个 B. 可能没有 C. 只能有多个 D. 可以1个或多个
- (5) 预处理命令是在()处理的。
A. 编译前 B. 编译后 C. 运行时 D. 连接时
- (6) typedef 语句的作用是()。
A. 定义一种结构类型
B. 定义一种结构类型的变量
C. 建立一种新的类型说明符
D. 给已有的类型定义新的类型名
- (7) 定义符号常量应选用()。
A. typedef 语句 B. #include 命令 C. #define 命令 D. 赋值表达式语句
- (8) 已知: #define SQ(x) x * x,在下列的各种宏替换中()是错误的。
A. 语句 m = SQ(5); 被替换为 m = 5 * 5;
B. 语句 m = SQ(a + b); 被替换为 m = (a + b) * (a + b);
C. 语句 m = SQ(a + 1) + 1; 被替换为 m = a + 1 * a + 1 + 1;
D. 语句 m = 5 / SQ(5); 被替换为 m = 5 / 5 * 5;
- (9) 下面对文件包含的描述中,()是错误的。
A. 被包含的文件可是文本文件也可以是二进制文件
B. 在一个源文件中可以包含多个.h 文件
C. 在一个被包含的.h 文件中还可包含另一个.h 文件
D. 系统提供的.h 文件,也可以使用双引号引用形式
- (10) 宏名的作用域是()。
A. 在定义它的文件中,从定义时起,到终止宏定义为止。
B. 在定义它的函数中,从定义时起,到终止宏定义为止。

- C. 在定义它的整个程序中,从定义时起,到程序结束时止。
D. 在定义它的分程序中,从定义时起,到该分程序结束时止。

2. 判断下述描述是否正确,对者划√,错者划×。

- (1) 简单宏定义是将一个标识符定义为一个字符串。
- (2) 宏定义的宏名必须大写,否则无效。
- (3) 宏定义的作用域是在定义它的函数内,并从定义时开始。
- (4) 宏定义只能放在函数体外,而不能写在函数体内。
- (5) 带参数的宏定义中,形参数只能是1个,不可有多个。
- (6) 文件包含命令中,只能包含扩展名为.h的文件。
- (7) 一个源程序可以使用多个文件包含命令,但是,一个文件包含命令只能包含一个文件。
- (8) 使用条件编译往往会增加目标代码的长度。
- (9) typedef 是一个语句,行尾应用分号结束。
- (10) 类型定义不是用来定义新的类型而是对已有类型再定义一个新的类型名。

3. 分析下列程序的输出结果。

(1)

```
#define M 1.6
#define a(a) M * a
main()
{
    int x,y;
    x=5;
    y=6;
    printf("%.1f\n",a(x+y));
}
```

(2)

```
#define XY(x,y) 3 * (x) + 5 * (y)
main()
{
    int x=5;
    float y=2.4;
    printf("%.2f\n",4/XY(x,y));
}
```

(3)

```
#define PR(a) printf("%d\t", (int)(a));
#define PRINT(a) PR(a);printf("ok!")
main()
{
    int i,a=5;
    for(i=1;i<=5;i++)
        PRINT(a-i);
    printf("\n");
}
```

(4)

```

main()
{
    int b=5;
    #define b 2
    #define f(x) b*(x)
    int y=3;
    printf("%d\n",f(y+1));
    #undef b
    printf("%d\n",f(y+1));
    #define b 3
    printf("%d\n",f(y+1));
}

```

(5)

```

#include <stdio.h>
#include "print1.h"
#define MAX(a,b) (a)<(b)?(b):(a)
main()
{
    int x=1,y=2;
    PRINT3(MAX(x++,y),x,y);
    PRINT3(MAX(x++,y),x,y);
}

```

print1.h 文件内容如下:

```

#define PR(a) printf("a=%d\t",(int)(a))
#define PRINT1(a) PR(a);putchar('\n')
#define PRINT2(a,b) PR(a);PRINT1(b)
#define PRINT3(a,b,c) PR(a);PRINT2(b,c)

```

(6)

```

#include "ff1.c"
main()
{
    int a,b;
    a=5;
    b=ff1(a);
    printf("%d\n",b);
}

```

ff1.c 的内容如下:

```

#define M(m) m*m
ff1(x)
int x;
{
    int a=3;
    return (-M(x+a));
}

```

```
}
```

4. 指出下列程序段中的错误。

(1)

```
#define ab(x, y) x * y + x / y
#include "ab. h" "xy. h"
#include <math. h>;
    :
```

(2)

```
typedef INT A, B
main()
{
    #define N 5;
    :
}
```

(3)

某程序内容如下:

```
main()
{
    :
    printf("%d\n", A);
    #include "ab. h";
    printf("%d\n", B);
    :
}
```

ab. h 文件内容如下:

```
#defin A 10
#defin B 20
```

(4) main()

```
{
    int n=5;
    :
    #if n
        printf("n is true.");
    #else
        printf("n is false.");
    #endif
    :
}
```

(5) 已知文件1包含文件2,而文件2中又要用到文件3的内容,则文件1中可以这样定义:

```
#include "file2. h"
#include "file3. h"
    :
```

其中, file2. h 和 file3. h 分别表示文件2和文件3的内容。

5. 按下列要求编程

(1) 写出下列程序段中, PRN3的宏定义命令。

```
:
```

```

float x, y, z;
x=0.11;
y=0.22;
z=0.33;
:
PRN3(x, y, z);
:

```

输出结果为:

x, y, z has value 0.1, 0.2, 0.3

(2). 写出下列程序中,A,B,PRN 的宏定义命令。

程序内容如下:

```

main()
{
    int i;
    char a[]—" * * * * * ";
    static int b[]={5, 6, 7};
    PRN("%2c", A, a);
    PRN("%3d", B, b);
    PRN("%2c", A, a);
}

```

该程序输出结果如下:

```

 *  *  *  *  *
 5      6      7
 *  *  *  *  *

```

第七章 指 针

指针是C语言中一个十分重要的特点。用好指针会提高C语言程序的效率,指针用得不好会带来很多麻烦。因此,掌握好和运用好指针是学习好C语言的关键。本章从讲解指针的概念入手,进而讲述指针的应用。搞清指针的概念是学好指针应用的基础。指针概念包括什么是指针、指针的表示、指针的赋值和指针的运算;指针的应用主要是指针在数组方面和函数方面的应用,关于指针其他方面的应用,如结构方面、文件方面等,将在后面章节中讲述。

7.1 指针的概念

搞清指针概念是学好用好指针的前提,在这一节中,主要讲述什么是指针、指针的表示和指针的赋值。

7.1.1 什么是指针

指针是一种特殊的变量。它的特殊性表现在哪些地方呢?由于指针是一种变量,它就应该具有变量的三要素:名字、类型和值。于是指针的特殊性就应表现在这三个要素上。指针的名字与一般变量的规定相同,没有什么特殊的地方。指针的值是某个变量的地址值,因此我们说指针是用来存放某个变量地址值的变量。指针的值与一般变量的值是不同的,这是指针一个特点。这就是说,指针是用来存放某个变量的地址值的,当然被存放地址值的那个变量是已经定义过的,并且被分配了确定的内存地址值的。一个指针存放了哪个变量的地址值,就说该指针指向那个变量。指针的第二个特点就表现在它的类型上,指针的类型是该指针所指向的变量的类型,而不是指针本身值的类型,因为指针本身值是内存的地址值,其类型自然是int型或long型。而指针的类型是由它所指向的变量的类型决定。由于指针可以指向任何一种类型的变量(C语言中所允许的变量类型),因此,指针的类型是很多的,例如,int型、char型、float型、数组类型、结构类型、联合类型,还可以指向函数、文件和指针等。

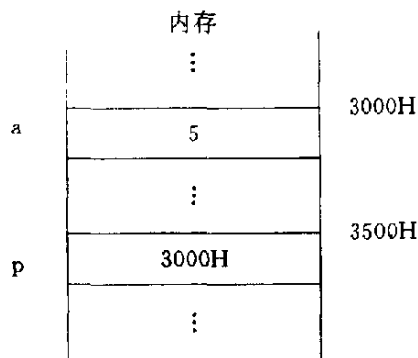
下面通过一个例子,进一步对指针的两个特点加深理解。例如,

```
int a=5, *p;
```

这里,说明了变量a是int型的,并且赋了初值。在说明语句中,*p表示p是一个指针,星号("*")是说明符,它说明后面的变量不是一般变量,而是指针,并且p是一个int型指针,意味着p所指向的变量是一个int型变量。假定,要指针p指向变量a,由于指针是用来存放变量的地址值的,因此,要将变量a的地址赋给指针p,变量a的地址表示为&a,这里&是运算符,表示取其后面变量的地址值。如果有

```
p=&a;
```

则p是指向变量a的指针。假定a被分配的内存地址是3000H,p和a的关系如下图所示:



图中标明变量 *a* 的内存地址为 3000H, 变量 *p* 的内存地址为 3500H, 变量 *a* 的值(即内容)为 5, 而指针 *p* 的值为 3000H, 可见指针 *p* 是用来存放变量 *a* 的地址值的。

C 语言中关于地址值的表示有如下规定:

- (1) 一般变量的地址值用变量名前加运算符 `&` 表示。例如, 变量 *x* 的地址值为 `&x` 等。
- (2) 数组的地址值可用数组名表示, 数组名表示该数组的首元素的地址值。数值中某个元素的地址值用 `&` 运算符加上数组元素名。例如:

```
int a[10], *p1, *p2;
p1=a;
p2=&a[5];
```

这里, `*p1` 和 `*p2` 是指向 `int` 型变量的指针, `p1=a`; 表示指针 *p1* 指向 *a* 数组的首元素; `p2=&a[5]`; 表示指针 *p2* 指向数组 *a* 的数组元素 *a[5]* 的指针。

- (3) 函数的地址值用该函数的函数名来表示, 指向函数的指针可用它所指向的函数名来赋值。

- (4) 结构变量的指针用 `&` 运算符加结构变量名来表示, 结构变量的成员的地址也用 `&` 运算符加结构变量的成员名来表示。关于结构变量和结构变量的成员将在“结构”一章中讲解。

关于 `&` 运算符的用法需要注意的是它可以作用在一般变量名前、数组元素名前、结构变量名前和结构成员名前等, 而不能作用在数组名前, 也不能作用在表达式前和常量前。

综上所述, 对指针的含意应作如下理解: 指针是一种不同于一般变量的特殊变量, 它是用来存放某个变量的地址值的, 它存放哪个变量的地址就称它是指向那个变量的指针。指针的类型不是它本身值的类型, 而是它所指向的变量的类型。简单地说, 对指针应记住如下两点:

- (1) 指针的值是地址值。
- (2) 指针的类型是它的所指向变量的类型。

7.1.2 指针的表示

在明确了指针的含意以后, 接着要学会正确地表示指针。由于指针具有不同的类型, 因此, 它在表示上也有差别。应记住各种不同类型指针的表示。

- (1) 指向基本类型变量的指针表示如下:

- ① 指向 `int` 型变量的指针, 例如:

```
int *p1, *p2;
```

这里, *p1* 和 *p2* 是两个指向整型变量的指针, *p1* 和 *p2* 前的 `*` 是表示指针的说明符。

- ② 指向 `char` 型变量的指针, 例如:

```
char * pc1, * pc2;
```

这里,pc1和 pc2是两个指向字符型变量的指针。

③ 指向 float 型变量的指针,例如:

```
float * pf1, * pf2;
```

```
double * pd1, * pd2;
```

这里,pf1和 pf2是两个指向单精度浮点型变量的指针。pd1和 pd2是两个指向双精度浮点型变量的指针。

(2) 指向数组的指针表示如下:

① 一般地,认为指向数组的指针就是指向该数组首元素的指针,例如:

```
int a[5][3],(* pa)[3];
```

其中,a 是一个二维数组的数组名,pa 是一个指向数组的指针名.pa 是一个指向每列有3个元素的二维数组的指针。例如,

```
pa=a;
```

则表示指针 pa 指向二维数组 a。

指向数组的指针的表示与指针数组的表示很相似,使用时要注意其区别。例如:

```
float m[3][2],* p1[3],(* p2)[2];
```

这里,m 是一个二维数组名,p1是一个一维一级指针数组名。所谓指针数组就是数组的元素为指针的数组.p1是指针数组名,数组 p1有3个元素,每个元素是一个一级指针,该指针指向 float 型变量.p2是一个指向数组的指针,它指向一个每列有2个元素的二维数组。可见,p1和 p2的表示形式很相似,前者是指针数组,后者是向数组的指针,其含意是完全不同的。

② 指向数组元素的指针一般是指向该数组的任何一个元素。例如:

```
float n[10][5],* p;
```

```
p=&n[5][1];
```

这里,p 是一个指向 float 型变量的指针,将数组 n 的某个元素的地址值,如 &n[5][1],赋给该指针 p,则 p 便是一个指向数组 n 的某个元素的指针。一般地,指向数组元素的指针是一个指向该数组元素所具有的类型的地变量的指针,它与指向数组的指针在表示上是有区别的。

(3) 指向函数的指针表示如下:

```
int (* pf)();
```

这里,pf 是一个指向函数的指针,它所指向的函数的返回值为 int 型数。

指向函数的指针与指针函数在表示上要区别开。下面的 pf 是一个指针函数:

```
int * pf();
```

这里,pf 是一个返回值为 int 型数的指针函数。所谓指针函数是一种返回值为指针的函数。

(4) 指向指针的指针表示如下:

```
int ** pp;
```

这里,pp 是一个指向指针的指针,即是一个二级指针。所谓二级指针是指它所存放的地址是一个一级指针的地址值,即它所指向的是一个一级指针。所以,二级指针又称为指针的指针。同样,三级指针所存放的是一个二级指针的地址值,该二级指针又存放着一级指针的地址值,该一级指针才存放某个变量的地址值。所以,三级指针是指针的指针的指针。依此类推。

关于结构变量的指针和文件指针以后再讲解。

7.1.3 指针的赋值

前面已经讲过,指针的值是地址值,因此,给指针赋值或赋初值要是一个地址值。各种变量的地址值的表示方法前面已经讲过了。所以给指针赋值不是一件困难的事情。

1. 赋值和赋初值

给指针可以赋值,也可以赋初值。

赋值是用一个赋值表达式语句进行;赋初值是在说明或定义指针的同时给它赋值。

不论是赋值还是赋初值,对一般的指针都是给予一个相对应的地址值;对于指针数组,则按其数组的赋值或赋初值方法进行赋值。例如:

```
int x, *p=&x;
```

这是给指向 int 型变量指针 p 赋初值,即将 int 型变量 x 的地址值赋给了 p。如果写成赋值的方式,如下所示:

```
int x, *p;  
p=&x;
```

又例如,

```
float y[2][3], *py[2]={y[0],y[1]};
```

这是给一个指针数组 py 赋初值,py 是一个具有 2 个元素的指针数组,它的每个元素是一个 float 型的指针,这里 y[0]和 y[1]是用来表示二维数组 y 的两个行地址,即将数组 y 看成是二行三列的一个数组,每一行是一个一维数组,它由 3 个元素组成。由此可见,一个二维数组可以表示为一个一维的指针数组,该指针数组的每个元素对应二维数组的每一行。上例又可写成赋值的形式,如下所示:

```
float y[2][3], *py[2];  
py[0]=y[0];  
py[1]=y[1];
```

或者,

```
int i;  
float y[2][3], *py[2];  
for(i=0; i<2; i++)  
    py[i]=y[i];
```

对于一个指向函数的指针,可用函数名给它赋值。例如:

```
double sin(),(*p1)();  
p1=sin;
```

这里,p1 是一个指向函数的指针,将一个函数名 sin 赋给了 p1,则 p1 便是一个指向函数 sin 的指针。

对于指向数组的指针,一般用相应的数组名给它赋值(或赋初值)。例如:

```
int x[3][5],(*px)[5];  
px=x;
```

这里,px 是一个指向数组的指针名,它所指向的数组是一个具有每列 5 个元素的二维数

组。将数组 x 的数组名赋给 px , 则 px 将指向二维数组 x 。

对于指向数组元素的指针, 就将它所指向的某个数组元素的地址值赋给它。

在对多级指针赋值时, 要注意所赋予的指针的级别要与所赋给的地址值相对应。这就是说, 给一个一级指针赋值, 则用一个变量的地址值就可以了; 给一个二级指针赋值, 则要用某个变量的地址的地址值; 依次类推。

关于其他类型指针的赋值或赋初值后面还会讲到。

给指针赋值还有一种常用的方法, 这就是使用存储管理函数 `malloc()`, 该函数的格式如下:

```
(void *)malloc(size)
int size;
```

该函数是用来分配内存地址的。该函数有一个参数 `size`, 用来表示所申请内存大小的字节数。该函数所分配的内存地址值, 可用来存放所指定的任何类型变量的地址值。该函数如果成功地分配了内存地址, 则返回一个地址值。否则, 返回 `NULL` (即0地址), 表示申请分配内存失败。例如:

```
char *s;
s=(char *)malloc(10*sizeof(char));
```

其中, s 是一个 `char` 型指针, 它可以用来存放一个字符串。通过调用系统提供的 `malloc()` 函数, 申请 `10 * sizeof(char)` 个字节的内存空间, 如果申请成功, s 将获得一个地址, 该地址值是内存中某个空间的首地址值。如果 s 的值为 `NULL`, 这说明申请失败, s 没有获得内存空间。又例如:

```
int *pa;
pa=(int *)malloc (sizeof(int));
```

pa 是一个指向 `int` 型变量的指针, 使用 `malloc()` 函数将获得一个内存地址值, 于是 pa 便是一个被赋了地址值的指针。

在实际应用中, 常用 `malloc()` 函数给指针赋值, 读者一定要掌握这种方法。

2. 指针赋值时应注意的事项

给指针赋值(或赋初值)除了要用地址值外, 还要注意下面几点。

(1) 指针被定义后, 只有赋了值(或赋了初值)才能使用。或者说, 没有被赋值的指针不能使用。使用没有被赋值的指针是很危险的, 有可能造成系统的瘫痪。道理是很简单的。因为一个没有被赋值的指针, 定义后它将被分配一个内存空间, 该空间仍保存着原来的内容, 即存在一个“无效”的地址值, 该地址值可能是内存中存放关键系统软件的地址, 一旦使用了该指针去改变了它所指向的内容, 则会造成系统软件被改变, 因此有可能造成系统的瘫痪。所以, 一定要记住, 指针在使用前一定要先赋值。

(2) 给指针赋值时一定要注意类型的一致。这就是说, `int` 型变量指针要赋一个 `int` 型变量的地址值, 或用 `malloc()` 函数赋值时, 前面要强制 `int` 型指针类型。例如:

```
int a, *pa;
pa=&a;
```

或者,

```
pa=(int *)malloc(sizeof(int));
```

都是正确的。而下列赋值是错误的，

```
int * pa;  
float b;  
pa=&b;
```

或者，

```
pa=(int *)malloc (sizeof(float));
```

(3) 可将一个已赋值的指针值赋给另一个同类型的指针。这里，有两点要注意：一个已赋值的指针，而没有被赋值的指针不能赋给另一个指针；二是同类型的指针，而不同类型的指针是不能这样赋值的。例如：

```
int a, * p, * q  
p=&a;  
q=p;
```

这里，先给指针 p 赋值，然后再将指针 p 赋给同类型的指针 q，于是指针 q 和 p 同时指向变量 a。

(4) 暂时不用的指针可以赋值 NULL。例如：

```
int * p;  
p=NULL;
```

其中，p 是一个暂时不用的指针被赋值为 NULL。前面讲过，指针不赋值就使用是很危险的。因此，为了避免这种危险，可将暂时不用的指针赋值 NULL，将来使用时再重新赋值。这样，被赋值为 NULL 的指针一旦被使用也不会带来危险。被赋值为 NULL 的指针又称为无效指针。

(5) 指针也可以被赋一个整型数值，但是使用这种赋值的方法要十分慎重。一般对于内存的地址分配情况不是十分清楚的人，请一定不要作这种冒险的事情。因为这种赋值方法，也会带来系统瘫痪的危险。

7.1.4 指针所指向变量的值

指针的值，前面已经讨论过了，它是所指向变量的地址值。而指针所指向变量的值就是该指针所存放地址的那个变量的值。例如：

```
int a=5, * p=&a;
```

显然，p 的值为变量 a 的地址值，而 p 所指向变量的值便是 a 的值，即 5。

变量的值可以直接地用变量来表示，如上例中，变量 a 的值是 5。而变量的值也可以间接地用指向该变量的指针来表示，即为该指针所指向的变量值。反过来说，指针可以间接地表示它所指向的变量的值。如何表示呢？这要使用单目运算符 *。* 作为单目运算符是用来表示指针所指向变量的值，或者称为指针所指向的内容。在上例中，*p 的值便是 5。因为 *p 表示 p 所指向变量的值。运算符 * 的功能是用来表示它后面的指针所指向的变量值。单目运算符 * 又称为“间接访问”运算符。

掌握指针这个概念，不仅要知道什么是指针，而且还要知道与指针相关的两个运算符，一个是单目运算符 &，另一个是单目运算符 *。简单地说，& 运算符是用来表示变量的地址值的；* 运算符是用来表示指针所指向变量的值。这两个运算符在指针的应用中经常遇到，一定要掌

握它们的功能和用法。

下面通过一个例子,进一步理解和掌握指针的值和类型,进一步搞清运算符 & 和 * 的功能和用法。

[例7.1] 分析下列程序的输出结果。

```
main()
{
    int x,y;
    int * px, * py, * p;
    px=&x;
    py=&y;
    p=(int *)malloc(sizeof(int));
    *px=5;
    *py=6;
    *p=7;
    printf("%p,%p,%p\n",px,py,p);
    printf("%p,%p\n",&x,px);
    printf("%d,%d,%d\n",*px,*py,*p);
    printf("%d,%d\n",y,*py);
}
```

执行该程序,输出结果如下:

```
地址值1,地址值2,地址值3
地址值1,地址值1
5,6,7
6,6
```

说明:

(1) 程序中先说明了三个指针 px,py 和 p,说明(或定义)指针时,在指针名前加 * 号。接着,用三个赋值表达式语句给定义的三个指针进行赋值,前两个分别将两个变量的地址值赋给了指针,后一个是使用 malloc() 函数进行分配内存单元的。赋了值的指针才可使用。

(2) 程序中又使用三个赋值表达式语句通过指针间接地给变量赋值。其中, * px=5; 与 x=5; 是等价的, * py=6; 与 y=6; 是等价的,这里使用的星号 * 是单目运算符。* px 表示指针 px 所指向的变量 x, * py 表示指针 py 所指向的变量 y, * p 表示 p 指针所指向的变量。通过指针可以间接地给它们指向的变量赋值。

(3) 程序中前两个 printf() 函数是用来输出指针的值和变量的地址值的。这里值得注意的是在 printf() 函数的控制串中使用了格式符 %p, %p 是 Turbo C 编译系统中提供的输出十六进制表示地址的格式符 p,这在标准 C 语言的输出格式中是没有的。使用 %d 格式输出地址值是可以的,但是在 16 位机器上,十进制整型数表示不了所有的地址,因而出现莫名其妙的负值,这是溢出后的结果,所以,在 Turbo C 编译系统中,常用 %p 来作输出地址值的格式符。第一个 printf() 函数的语句输出的是三个地址值,分别是指针 px 的值,指针 py 的值和指针 p 的值。第二个 printf() 函数的语句输出两个相同的地址值,指针 px 的值就是变量 x 的地址值 &x。

程序中后两个 printf() 函数输出的变量值,第三个 printf() 函数的语句输出三个指针所指向的变量值,即为 x,y 和指针 p 所指向变量的值。第四个 printf() 函数的语句输出两个变量

值,这两个变量值是相等的,因为它们是通过不同形式表示的同一个变量。 y 是直接表示的变量值, $*py$ 是间接表示的变量 y 的值。

7.2 指针的运算

由于指针是一种特殊的变量,它的运算也是很有限的。指针仅允许如下4种运算:

- 赋值运算
- 指针加减整数运算
- 两个指针相减运算
- 两个指针比较运算

指针的运算实际上是地址的运算,而又不同于地址运算。

本节讲述以上内容。

7.2.1 指针的赋值运算

指针可以被赋值,使用赋值表达式语句,或者用某个变量的地址值,或者用 `malloc()` 函数。

指针也可以被赋初值,其方法与一般变量相同,只是用地址值。

关于指针的不同类型的各种赋值方法在上一节都已讲到,这里不再重复。

7.2.2 指针加减整数运算

一个指针可以加上或减去一定范围内的一个整数,以此来改变指针的地址值。例如,

```
int a[10], *p=a;
```

p 是一个指向数组 a 的首元素的指针,而 $p+=1$ 则表示指针 p 指向了数组元素 $a[1]$,即指向了数组的下一个元素。 $p+=1$ 表达式与 $p++$ 表达式是等效的,都是用来将指针 p 所指向的数组元素向下移一个。同理, $p+=2$ 则表示将 p 所指向的元素向下移动二个,即使 p 指向 $a[2]$ 元素。指针 p 还可以作如下运算:

$p++$, $p--$, $p+=i$, $p-=i$, $p+i$, $p-i$ 等等。

这里需要注意的是:指针加1(即 $p++$)不是简单地将 p 的值加上1,而是将 p 的值加上1倍的它所指向的变量占用的内存字节数。指针加 i (即 $p+=i$)是将 p 的值加上 i 倍的它所指向变量占用的内存字节数。对于 `int` 型变量,它占用内存的字节数为2,因此, $p+=i$;实际上是 $p+=2*i$,对于指针指向的变量为 `float` 型的, $p+=i$,实际上是 $p+=4*i$,等等。对于指针减去一个整数也是如此。这将告诉我们,指针运算不同于地址运算,尽管它们实际上都是地址运算。

7.2.3 两个指针相减运算

两个指针可以相减,但是这里要有一定的条件,比如指向同一个数组的两个指针可以相减。因此,应该说:在一定条件下,两个指针可以相减,而不是说任意的两个指针都可以相减。实际上,任意的毫无关联的两个指针相减是没有意义的。而指向同一个数组的两个不同元素的指针相减,则表示两个指针相隔元素的个数。例如,有两个指针指向同一个字符串,其中一个指针指向字符串的首字符,另一个指针指向字符串的结束符,这两个指针相减的绝对值就是该字符串所具有的字符个数,即字符串长度。

[例7.2] 计算已知字符串的长度。

```
main()
{
    int i;
    char s[]="abc def";
    char *p1, *p2;
    i=0;
    p1=p2=s;
    while(s[i++]!='\0')
        p2++;
    printf("%d\n", p2-p1);
}
```

执行该程序,输出如下结果:

7

说明:该程序中,定义了两个字符型指针 p1 和 p2,开始被赋值,让这两个指针都指向字符型数组 s 的首元素。然后,通过 while 循环来移动 p2 指针,每执行一次循环体, p2 指针加1,即向下移动一个元素。循环条件是判断字符数组中何时为字符串结束符。第一次循环,取 s[0] 元素,这时 p2 加1,第二次循环,取 s[1] 元素,则 p2 又加1,……当第7次循环结束后,再取出 s[7] 则为 '\0',即结束循环,这时 p2 已经加了7次1,所以输出 p2-p1 的值应该为7。因为 p1 一直没有改变,它仍指向 s[0] 元素。

7.2.4 两个指针比较的运算

在一定的条件下,两个指针可以进行比较运算,即可进行大于、小于、大于等于、小于等于、相等和不等的运算。任意的毫无关联的两个指针进行比较是毫无意义的。例如,指向同一个数组的两个指针可以进行比较。如果两个指向同一个数组的指针相等,则表示这两个指针是指向同一个元素,否则两个指针不等,表示这两个指针不是指向同一个元素,而是指向两个不相同的元素。

[例7.3] 把一个字符串中的字符逆序输出。

程序内容如下:

```
main()
{
    char s[]="abcdefgh";
    char *p;
    for(p=s+7; p+1!=s; p--)
        printf("%c", *p);
    printf("\n");
}
```

执行该程序输出结果如下:

hgfedcba

说明:程序中,定义了一个字符数组 s,又定义了一个 char 型指针 p。通过指针加上整数运算,先使指针 p 指向已知字符串的最后一个字符 h,再输出 p 所指向的字符,每次循环 p 指针

减1,判断 $p+1$ 与 s 是否相等, s 是数组名,即是指向数组首元素的指针。如果 $p+1$ 与 s 相等,则说明 p 将指向该数组首元素前边的一个元素,该元素实际是不存在的,这时退出 for 循环,并完成对该字符串的逆序输出。

该例中关于指针运算的应用有如下几点:

- (1) 指针赋值。 $p=s+7$;是一个赋值表达式语句,将 $s+7$ 的值赋给指针 p 。
- (2) 指针加减整数运算。 $p--$ 是指针减1表达式。
- (3) 指针比较运算。 $p+1!=s$ 是一个关系表达式,判断 $p+1$ 与 s 是否相等。

7.2.5 指针运算与地址运算的区别

指针运算实际上是地址运算,但它又不同于地址运算。关于这个问题实际上前面(7.2.2中)已经讲过。指针加1不是将指针的值加上1,而是将指针值加上1倍的该指针所指向变量占内存的字节数,在16位机上,int 型指针是占2个字节,float 型指针占4个字节,double 型指针占8个字节等等。而地址加1是加绝对地址值1。下面通过一个例子说明指针加1与地址加1的区别。如果 p 是一个指针,则 $p+1$ 表示指针加1;而 $(\text{int})p+1$ 则表示地址加1。

[例7.4] 分析指针加1与地址加1的区别。

分析该程序输出结果。

```
main()
{
    char c, *cp1=&c, *cp2;
    int i, *ip1=&i, *ip2;
    double d, *dp1=&d, *dp2;
    cp2=cp1+1;
    ip2=ip1+1;
    dp2=dp1+1;
    printf("%d,%d,%d\n",cp2-cp1,ip2-ip1,dp2-dp1);
    printf("%d,%d,%d\n",(\text{int})cp2-(\text{int})cp1,
        (\text{int})ip2-(\text{int})ip1,(\text{int})dp2-(\text{int})dp1);
}
```

执行该程序输出如下结果:

```
1,1,1
1,2,8
```

说明:该程序中,分别定义了二个 char 型指针、int 型指针和 double 型指针,并且分别给这些指针赋了值和赋了初值。然后,使用 printf() 函数分别输出指针运算的结果和地址运算的结果,其结果是不同的,这将说明指针运算和地址运算的区别。同时也将看出指针运算与地址运算表示上的不同。其中, $dp2-dp1$ 为指针运算,而 $(\text{int})dp2-(\text{int})dp1$ 为地址运算。前一个表达式的值为1,说明这两个指针相距为1个 double 型指针所指向变量占有的内存字节数,简称为1个单位字节数,对 double 型指针,该单位字节数为8。后一个表达式的值为8,说明这两个指针相距8个字节,因为机器内存是按字节编址的。这两种不同的表示形式实质是一样的。对 double 型指针来讲,指针相距为1就等于相距8个字节,因为一个 double 型变量占内存8个字节。

7.3 指针与数组

C 语言中,对数组元素的存取允许有两种方式,一种是下标方式,另一种是指针方式。但是,比较这两种方式还是尽量使用指针方式,虽然允许使用下标,由编译系统再将下标方式转换为指针方式。因此,为了提高效率还是应该尽量使用指针方式为好,免得浪费机器的转换时间。可是,由于在某种情况下,用下标表示比用指针表示更方便、更容易,使用下标表示也是可以的。

7.3.1 数组名是一个常量指针

为使数组更加方便地用指针表示,C 语言中规定数组名是一个指针,该指针是指向该数组的首元素,因此,数组名是一个常量指针。

所谓常量指针是指一种其值不能改变的指针。常量指针与变量指针是不同的,其区别就在于变量指针是可以改变的,一般定义的指针都是变量指针。例如,

```
int a[10], *p=a;
```

这里,a 是数组名,是常量指针,p 是指针名,是变量指针。p++,p--,p+=5,p-=5 等运算都是合法的,而 a++,a--,a+=5,a-=5 等运算都是非法的。p=a+2 运算是合法的。这里,要区分常量指针和变量指针的不同。

[例7.5] 求一个指定字符串的长度。

前面编写过求字符串长度的程序。下面用一种更简洁的方法编写。

程序内容如下:

```
main()
{
    char s[]="abcdef", *p=s;
    while(*p)
        p++;
    printf("The string length is %d\n",p-s);
}
```

执行该程序输出如下结果:

This string length is 6

说明:该程序中,s 是数组名,p 是指向 s 数组的指针。通过循环(while)将指针 p 由指向 s 数组首元素开始,一直向后移动到指向字符串结束符 '\0' 为止。显然,p-s 便是字符串的长度。

[例7.6] 编程对一个已知字符串作逆序输出。该题目前面已经作过,下面再用一种方法编程。

程序内容如下:

```
#include <stdio.h>
main()
{
    char s[]="abcdefgh";
```

```

char *ch=&s[8];
while( --ch>=s)
    putchar(*ch);
    putchar('\n');
}

```

执行该程序输出结果如下：

hgfedcba

说明：该程序中，s 是数组名，ch 是一个指向 s 数组第8个元素的指针，即指向该字符串的结束符。(记住：数组元素是从第0个数起!)通过 while 循环，ch 指针每次减1，通过 putchar() 函数将字符串从最后一个字符开始输出，形成逆序输出直到该字符串首字符被输出为止。

7.3.2 数组元素的指针表示

数组元素可以用指针表示，也可以用下标表示，而用指针表示比用下标表示可提高速度。因此，尽量采用指针表示。

下面讨论一维数组、二维数组和三维数组的元素的各种表示方法。

1. 一维数组

假定 a 是一个一维数组，其定义格式如下：

```
int a[10];
```

该数组元素的下标表示为：

$a[i]$, $i=0, 1, \dots, 9$ 。

该数组元素的指针表示为：

$*(a+i)$, $i=0, 1, \dots, 9$ 。

因为 a 为该数组的首元素地址， $a+i$ 为该数组第 i 个元素的地址，而 $*(a+i)$ 则为该数组的第 i 个元素的值。

一维数组的指针表示很简单，使用数组名(实际上指向数组首元素的指针)可以很方便地表示出数组各元素的值，实际上数组是按一定顺序存放在内存，由于各元素类型相同，即每个元素占用的内存单元的字节数也相同，因此，只是知道首元素的地址，便可通过指针加1办法来访问数组的所有元素。

[例7.7] 分析下列程序输出结果。

```

main()
{
    static int a[]={0,1,2,3};
    int *p=a;
    printf("%d,%d,%d\n",a[1],p[2],*(p+3));
}

```

执行该程序输出结果如下：

1,2,3

说明：程序中，a 是一个一维数组名。 $a[1]$ 是该数组的第1个元素(数组元素是从0数起)。p 是一个指向数组 a 首元素的指针， $p[2]$ 表示与 $*(p+2)$ 是等价的，即表示 $a[2]$ 元素，而 $*(p+3)$ 表示了 $a[3]$ 元素。因此，有上述结果。

[例7.8] 分析下列程序的输出结果。

```
main()
{
    static int a[] = {1,3,5,7,9};
    int i, *p;
    for(p=a, i=0; p+i <= a+4; p++, i++)
        printf("%4d", *(p+i));
    printf("\n");
    for(p=a+4, i=0; i<5; i++)
        printf("%4d", p[-i]);
    printf("\n");
}
```

执行该程序输出结果如下：

```
1  3  5  7  9
9  7  5  3  1
```

说明：该程序中，a 是一维数组名，p 是指向数组 a 的指针。第一个 for 循环时，p 指向 a 数组的 a[0] 元素，第二个 for 循环时，p 指向 a 数组的 a[5] 元素。

第一个 for 循环将顺序输出 a 数组中的某些元素，在 printf() 函数中，使用 *(p+i) 来代替数组 a 的各元素值，它等价于 a[i]。

第二个 for 循环中，printf() 函数的参数是 p[-i]，该参数等价于 *(p-i)，p 开始时指向数组 a 的 a[5] 元素，i 由 0 逐次增加到 5，于是按其相反顺序输出 a 数组中 5 个元素。

[例7.9] 编程把一个 int 型数组中的若干个元素按相反顺序存放。

```
#define N 8
int b[N] = {8,7,6,5,4,3,2,1};
main()
{
    int i, n, j, t;
    printf("The original array:\n");
    for(i=0; i<N; i++)
        printf("%4d", *(b+i));
    printf("\n");
    n = (N-1)/2;
    for(i=0; i<=n; i++)
    {
        j = N-1-i;
        t = b[i];
        b[i] = b[j];
        b[j] = t;
    }
    printf("The array has been inverted:\n");
    for(i=0; i<N; i++)
        printf("%4d", *(b+i));
    printf("\n");
}
```

执行该程序输出结果如下:

```
the original array:
8 7 6 5 4 3 2 1
the array has been inverted:
1 2 3 4 5 6 7 8
```

说明:该程序中,定义一个符号常量 N,用来表示数组的大小,改变数组大小时只要改变 N 的定义值即可。数组 b 是一个一维的外部数组,并赋了初值。程序中先输出了原数组的元素顺序。接着,通过 for 循环,从两头向中间顺序交换两个元素,先将第 0 个元素与最末一个元素交换,再将第 1 个元素与倒数第 2 个元素等,一直到中间。于是,使原数组中存放的数据改变了顺序。最后,又用 for 循环输出反序的数据。这里,所用的数组元素有下标表示的,也有指针表示的。最好都用指针表示,可提高速度。

2. 二维数组

假定 b 是一个二维数组,它的定义格式如下:

```
int b[3][5];
```

该数组的下标表示如下所示:

```
b[i][j], i=0, 1, 2; j=0, 1, 2, 3, 4。
```

该数组按其在内存中的存放顺序,用一级指针表示如下:

```
*(&b[0][0]+5*i+j)
```

其中,i 和 j 取值同上。 $\&b[0][0]$ 为该数组的首元素地址值。由于二维数组存放在内存中的顺序是先存放第 0 行的第 0 列到第 4 列共 5 个元素,接着存放第 1 行的第 0 列到第 4 列的 5 个元素,最后存放第 2 行的第 0 列到第 4 列的 5 个元素。共存放 15 个元素。开始的地址为 $\&b[0][0]$ 。因此, $\&b[0][0]+5*i+j$ 表示了第 i 行第 j 列元素在内存中的地址,而 $*(&b[0][0]+5*i+j)$ 表示第 i 行第 j 列元素的值。

这里要注意,不能将上述表示式写成这样:

```
*(b+5*i+j)
```

这里的 b 是数组名,但在二维数组(三维数组也一样)中数组名表示第 0 行元素的行地址,尽管它的值与该数组的首元素地址值相等,但是,它的含意与首元素地址不同,它是一个行地址,这一点后面还会讲到。

二维数组除了上述的两种表示之外,还有如下几种表示。

因为二维数组可以看成是由行和列组成的,例如, `int b[3][5];` 中的二维数组 b 可以看成是三行五列。b 是一个数组名,b 数组包含有 3 个元素: `b[0]`, `b[1]`, `b[2]`。每个元素实际上表示一行,它又是一个一维数组,它包含 5 个元素,每个元素是一个列元素。例如, `b[0]` 所代表的一维数组的 5 个元素分别为: `b[0][0]`, `b[0][1]`, `b[0][2]`, `b[0][3]` 和 `b[0][4]`。

在二维数组中, `b` 代表第 0 行首地址,而 `b[0]` 是一个一维数组(行数组)的名字,它代表该行的首列地址。因此, `b[0]` 代表第 0 行中第 0 列元素的地址。同理, `b+1` 代表第 1 行的首地址,而 `b[1]` 代表第 1 行第 0 列元素的地址; `b+2` 代表第 2 行的首地址, `b[2]` 代表第 2 行第 0 列元素的地址。在这里, `b+i` 与 `b[i]` 所代表的含意是不同的,但它们的值是相等的。 `b+i` 与 $\&b[i]$ 是等价的; `b[i]` 是 $\&b[i][0]$ 是等价的。这里还需要注意的是:尽管 `b[i]` 可认为它代表一维数组名,但它本身

并不占有实际的内存空间,当然也不存放它的各个元素值,它只有一个与 $\&b[i][0]$ 相同的地址值。

综上所述,一个二维数组可认为是由若干个行(每行看作是一维数组)组成的,每行中又含有若干个列。例如, $b[3][5]$ 可看成为由3个元素的行数组和5个元素的列数组组成。于是二维数组的表示可用如下形式:

$*(b[i]+j)$

其中, $b[i]$ 是代表第 i 行的首地址, $b[i]+j$ 代表第 i 行第 j 列的地址,因此, $*(b[i]+j)$ 表示第 i 行第 j 列元素。

$(*(b+i))[j]$

其中, $b+i$ 代表了第 i 行首地址, $*(b+i)$ 等价于 $b[i]$ 代表了第 i 行的数组名,而 $(*(b+i))[j]$ 表示了该数组第 j 个元素,即表示第 i 行第 j 列元素。

$*(*(b+i)+j)$

其中, $*(b+i)$ 等价于 $b[i]$ 代表了第 i 行第0列的地址, $*(b+i)+j$ 代表第 i 行第 j 列的地址,因此, $*(*(b+i)+j)$ 代表第 i 行第 j 列的元素值。二维数组元素的这种表示,实际上是二级指针的表示方法。

综上所述,将二维数组的元素各种表示归结如下:

下标表示: $b[i][j]$

一级指针表示: $*(\&b[0][0]+5*i+j)$

行用下标列用指针表示: $*(b[i]+j)$

行用指针列用下标表示: $(*(b+i))[j]$

二级指针表示: $*(*(b+i)+j)$

【例7.10】 分析下列程序输出结果,进一步加深对二维数组的各种表示的理解。

```
main()
{
    static int a[2][3]={1,3,5,7,9,11};
    printf("%d,%d,%d\n",a,*a,* *a);
    printf("%d,%d,%d\n",a[0],&a[0],*(a+0));
    printf("%d,%d,%d\n",a[1],a+1,*(a+1));
    printf("%d,%d,%d\n",&a[1],* *(a+1),(*(a+1))[0]);
    printf("%d,%d,%d\n",a[1][1],* (*(a+1)+1),*(a[1]+1));
}
```

执行该程序输出结果如下:

```
404,404,1
404,404,404
410,410,410
410,7,7
9,9,9
```

说明:

(1) 输出结果中,404和410是地址值,其余的为元素值。

(2) $&a[i]$ 与 $a[i]$ 的区别。 $&a[i]$ 等价于 $a+i$,代表第 i 行的首地址,它是针对行的; $a[i]$ 等价于 $*(a+i)$,也等价于 $*(a+i)+0$,代表第 i 行第0列元素的地址, $a[i]$ 是数组名,它的元素是第 i 行的各列元素, $a[i]$ 是针对列的。实际中, $&a[i]$ 与 $a[i]$ 的值是相等的。这里,不能把 $&a[i]$ 认为是 $a[i]$ 的地址,因为实际上 $a[i]$ 是不存在的,只是一种表示方法,用来代表第 i 行首列地址。 $a[i]$ 的值是第 i 行第0列元素的地址值。而 $&a[i]$ 也是一种表示方法,代表第 i 行的首地址,它的值与第 i 行第0列元素的地址值相同。同样道理,也不能认为 $a+i$ 的单元内容是 $*(a+i)$ 。

程序中, a 等价 $a+0$, $*a$ 等价 $*(a+0)$, $a[0]$ 等于 $*(a+0)$, $&a[0]$ 等价 $a+0$, $*(a+0)$ 等价于 $*a$ 。它们值都是相同的,但是含意并不都相同。其 a 和 $&a[0]$ 是等价的, $*a$ 和 $a[0]$ 和 $*(a+0)$ 是等价的。还有 $a[1]$, $a+1$, $*(a+1)$ 和 $&a[1]$ 的值是相同的,但是它们的含意并不都相同。其中, $&a[1]$ 等价于 $a+1$,而 $a[1]$ 等价于 $*(a+1)$ 。

(3) 该程序中,代表数组元素值的表达式有如下这些:

$*a$ 等价于 $*(*(a+0)+0)$,即是数组元素 $a[0][0]$,其值为1。

$*(a+1)$ 等价于 $*(*(a+1)+0)$,即是数组元素 $a[1][0]$,其值为7。

$((a+1))[0]$ 即是数组元素 $a[1][0]$ 的行用指针列用数组的表示形式,其值为7。

$a[1][1]$, $*(*(a+1)+1)$ 和 $*(a[1]+1)$ 都等价于 $a[1][1]$,其值都为9。

[例7.11] 分析下列程序的输出结果。

```
int b[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
main()
{
    int i,j;
    for(i=0;printf("\n"),i<3;i++)
        for(j=0;j<4;j++)
            printf("%5d",*(*(b+i)+j));
    printf("%5d %5d%5d%5d\n",***(b+1),*(*(b+2),
        *(b[2]+3),(*(b+1))[2]);
}
```

执行该程序输出如下结果:

1	2	3	4
5	6	7	8
9	10	11	12
5	3	12	7

说明:该程序出现了5处二维数组元素的不同表示形式。其中, $*(*(b+i)+j)$ 相当于 $b[i][j]$, $*** (b+1)$ 相当于 $b[1][0]$, $*(*(b+2)$ 相当于 $b[0][2]$, $*(b[2]+3)$ 相当于 $b[2][3]$, $((b+1))[2]$ 相当于 $b[1][2]$ 。因此,会有上述结果。

[例7.12] 假定有5个学生,每人有3门功课的成绩,编程输出每个学生的平均分数。编程如下:

```
float score[5][3]={{86,90,78},{86,84,96},{91,76,88},
    {90,77,66},{78,97,68}};
main()
{
    int i,j;
```

```

float aver[5];
for(i=0;i<5;i++)
    aver[i]=0.0;
for(i=0;i<5;i++)
{
    for(j=0;j<3;j++)
        *(aver+i) += *(*(score+i)+j);
    aver[i]/=3;
}
for(i=0;i<5;i++)
    printf(" %8.2f", *(aver+i));
printf("\n");
}

```

执行该程序输出如下结果:

84.67 88.67 85.00 77.67 81.00

说明: 该程序中出现的数组元素使用了两种表示, 一种是下标表示, 另一种是指针表示。

3. 三维数组

假定 c 是一个三维数组, 它的定义格式如下:

```
int c[3][5][7];
```

该数组的下标表示如下:

$c[i][j][k]$, $i=0, 1, 2$; $j=0, 1, 2, 3, 4$; $k=0, 1, \dots, 6$ 。

该数组按其在内存中的存放顺序, 用一级指针表示如下:

$*(\&c[0][0][0]+5*7*i+7*j+k)$

其中, i, j, k 取值同上。 $\&c[0][0][0]$ 为该数组的首元素地址。由于三维数组存放在内存中的顺序是 i 和 j 为 0, k 由 0 逐次增 1 直到 6, 然后, i 为 0, j 为 1, k 再由 0 增至 6, 依次类推, 按照后面的维数的下标变化比前面快的原则进行存放。 i 增加 1 时, 实际上已增加 35 个元素; j 增加 1 时, 也增加了 7 个元素, 该数组共有 105 个元素。 $\&c[0][0][0]+5*7*i+7*j+k$ 表示该数组的 $c[i][j][k]$ 元素的地址值。取星号后便是该数组的 $c[i][j][k]$ 元素值了。

同样, 应注意的是 c 与 $\&c[0][0][0]$ 的含意不同。 c 是数组名, 它的值是该数组首元素地址, 即与 $\&c[0][0][0]$ 值相同, 但它实际上是表示第 0 个二维数组的首地址。即将三维数组看成是一个元素为二维数组的一维数组。该三维数组是一个由 3 个元素组成的一维数组, 而每个元素是一个 5 行 7 列的二维数组。 $c+i$ 表示第 i 个二维数组的首地址。 $\&c[0][0][0]$ 是该数组的首元素的地址。

三维数组除了上述两种表示方法之外, 还有如下的几种表示。对下述表示方法的理解可以将三维数组看成是数组元素为一维数组的二维数组。即将 $c[3][5][7]$ 看成为 3 行 5 列的二维数组, 每个元素是一个 7 个元素的一维数组, 关于一维数组和二维数组的表示前边都已讲过, 按照这种理解三维数组元素的表示方法如下:

为了描述方便, 将三维数组看成是由行、列和组构成, 即对上述的三维数组看成为 3 行 5 列 7 组。行、列、组都看成为一维数组。

行、列用下标, 组用指针表示:

$*(c[i][j]+k)$

行、组用下标,列用指针表示:

`(*(c[i]+j))[k]`

行用指针,列、组用下标表示:

`(*(c+i))[j][k]`

行、列用指针,组用下标表示:

`((*(c+i)+j))[k]`

行用下标,列、组用指针表示:

`*(*(c[i]+j)+k)`

行、组用指针,列用下标表示:

`*((*(c+i))[j]+k)`

行、列、组都用指针表示:

`*(*(c+i)+j)+k)`

三维数组元素共有上述9种表示方法。

下面给出三维数组的各种地址的表示方法:

行地址表示为: `c+i` 或 `&c[i]` 等形式。

行、列地址表示为: `*(c+i)+j`, `c[i]+j`, `&c[i][j]` 等形式。

行、列、组地址(即元素地址)表示为:

`*(*(c+i)+j)+k`, `*(c[i]+j)+k`, `&c[i][j][k]` 等形式。

[例7.13] 分析下列程序的输出结果。

```
int a[2][2][3]={{{1,2,3},{4,5,6}},{7,8,9},{10,11,12}}};
main()
{
    int i,j,k;
    a[0][0][1]=20;
    *(a[0][1]+2)=21;
    (*(a+1))[0][1]=22;
    (*(a[1]+1))[0]=23;
    (*(a[0]+1)+1)=24;
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
        {
            for(k=0;k<3;k++)
                printf("%4d",*(*(c+i)+j)+k);
            printf("\n");
        }
        printf("\n");
    }
    printf("\n");
}
```

执行该程序输出结果如下:

1 20 3

```

4    24    21
7    22    9
23   11    12

```

说明：这是一个练习三维数组元素的各种表示方式的程序，该程序中给出了三维数组的几种表示方式。如果二维数组的几种表示方式比较熟悉，该例题是不难理解的。

7.3.3 字符数组、字符指针和字符串处理函数

1. 字符数组和字符串

字符数组是指数组元素为字符的那种数组。字符数组也有一维字符数组、二维字符数组和三维字符数组等等。例如，

```

char s1[5];
char s2[3][4];
char s3[2][3][5];

```

其中，s1是一维字符数组名，它包含5个元素，每个元素是一个字符。s2是二维字符数组名，s3是三维字符数组名，它包含30个元素，每个元素是一个字符。

字符数组的赋值和赋初值的办法与数字数组相同。

在C语言中，最常用的字符数组是字符串，字符串是存放在一个字符数组中，存放一个字符串的字符数组是一维字符数组，而多维字符数组可以存放多个字符串。这里，需要搞清楚的一个概念是一维字符数组不等于字符串，但是字符串是一维字符数组。因为在一维字符数组中存放的若干个字符，如果该字符数组以 '\0' 为结束，则是字符串，否则不是字符串，而是一般字符数组。例如，

```

char s1[3]={'a','b','c'};
char s2[3]={'a','b','\0'};

```

其中，s2是一个字符串，而s1是一般的一维字符数组。给字符串赋初值时可将一个该字符串直接赋给一个字符数组。例如，

```

char s2[3]="ab";

```

这里，系统给s2自动添加一个字符串结束符 '\0'。

同样，给二维字符数组也可以用字符串赋初值。例如，

```

char ss[3][4]={"abc","mnp","xyz"};

```

ss是二维字符数组名，它包含有3个一维数组（每个一维数组4个元素），每个一维数组是一个字符串，该数组可存放3个不超过4个元素（含 '\0'）的字符串。

[例7.14] 字符串的输入和输出。

分析下列程序的输出结果。

```

#include <stdio.h>
main()
{
    char s1[]="abcde",s2[6];
    int i;
    for(i=0; i<5; i++)
        s2[i]='f'+i;

```

```

s2[i]='\0';
printf("s1: %s\ns2: ", s1);
puts(s2);
printf("%d\n", *s1);
putchar(*s2);
printf("\n");
}

```

执行该程序输出如下结果:

```

s1: abcde
s2: fghij
97
5

```

说明:

(1) 字符串输入。本程序中表明了可以给字符数组用一个字符串赋初值,当然也可以用初始值表的办法给字符数组赋初值,不要忘记加上字符串结束符 '\0'。本程序中也表明了可以给字符数组中各个元素赋值,即将一个字符赋给每个元素,该例中是通过 for 循环给字符数组 s2 赋值,每次循环将一个字符赋给 s2 的一个元素。最后将 '\0' 赋给 s2,使它成为一个存放字符串的字符数组。这里,要注意的是 s2 是 6 个元素,通过 for 循环给 s2 的前 5 个元素赋字符,而最后一个元素通过赋值表达式语句 s2[i]='\0'; 给它赋一个字符串结束符。

(2) 字符串输出。字符串输出使用 printf() 函数时,格式符用 %s,对应的表达式用要输出的字符串首字符的地址值。也可以使用 puts() 函数输出一个字符串。输出字符串的字符时,使用 printf() 函数其格式符用 %c 或 %d(用 %d 输出字符的 ASCII 码值),对应的表达式用存放要输出字符的数组元素。例如,*s1 则可表示 s1[0],对应字符的 ASCII 码值是 97。也可以使用 putchar() 函数来输出一个字符。

2. 字符指针

字符指针是一种专门用来指向字符串首字符的指针。C 语言中提供了字符指针,它可以更方便地对字符串进行处理。对于字符指针可以直接将一个字符串给它赋初值或赋值,这比使用一维字符数组方便多了。例如:

```

char *p1, *p2="abcd";
p1="mnpq";

```

可将一个字符串直接给字符指针赋初值,也可以赋值,这里 p1, p2 是两个字符指针。不要把 p1 和 p2 理解为字符串变量,把一个字符串赋给这个变量。而 p1 和 p2 是两个 char 型的指针,这里, p1="mnpq"; 不是把字符串存放在 p1 中,而是把字符串 "mnpq" 存放在内存的某个单元里,把该单元的首地址赋给字符指针 p1。实际上,这是用一个无名的字符数组来存放 "mnpq" 字符串。p1 只是指向该字符串首字符地址的一个指针。

[例 7.15] 编程将一个字符串每个字符加 1 后拷贝成另一个字符串中。将已知字符串赋给一个字符指针,每个字符加 1 后生成的新字符串存放在一个字符数组中。

程序内容如下:

```

main()
{

```



```

int i;
char *p1,a[16];
p1="I am a teacher.";
for(i=0;i<15;i++)
    a[i]=*p1++ + 1;
a[i]='\0';
printf("%s\n",a);
}

```

执行该程序输出结果如下：

```
J!bn!b!ufbdifs/
```

说明：程序中 p1 是一个字符指针，将一个已知字符串直接给它赋值，该指针指向字符串中首字符。使用 for 循环将 p1 所指的字符串的字符逐个加 1 后赋值给字符组 a 的对应元素。最后，在 a 数组中加以 '\0'，于是 a 数组中存放着一个字符串，该字符串是 p1 所指向的字符串每个字符加 1 后获得的字符串。可见，使用字符指针比使用字符数组更方便。读者可将该程序中的 a 数组也改写为字符指针。

【例 7.16】 分析下列程序，掌握字符指针的输出方法。

```

#include <stdio.h>
main()
{
    char *p, *q;
    p=q="HELLO";
    while(*p)
        putchar(*p++);
    putchar('\n');
    p=q;
    printf("%s\n", p);
    printf("%s\n", p+3);
}

```

执行该程序输出结果如下：

```
HELLO
HELLO
LO
```

说明：该例中给出了输出字符指针所指向的字符串的两种方法，请读者自己分析。

3. 字符串处理函数

C 语言为了方便对字符串的处理，提供了一些字符串处理函数。一般地，系统将字符串处理函数放在 string.h 文件中。

系统提供的常用的字符串处理函数有如下几种：

(1) 字符串长度函数 strlen()

该函数是用来计算某个已知字符串的长度的。字符串的长度是指字符串中有效字符的个数，它不包含字符串结束符 '\0'。

该函数格式如下：

```
strlen(s)
```

其中,参数 *s* 是存放字符串的字符数组名,或字符指针名以及字符常量。例如,

```
strlen("abcde");  
char *p="mnpq";  
strlen(p);
```

该函数返回一个 `int` 型数值,表示字符串的长度。上面例子中,`strlen("abcde");`返回值为5,`strlen(p)`返回值为4。

(2) 字符串比较函数 `strcmp()` 和 `strncmp()`

这两个函数的功能是对两个字符串参数进行比较。返回值为 `int` 型数值。返回值为0表示两个字符串相同,返回值不为0(大于0或小于0)则表示两个字符串不等。将两个字符串从头到尾顺序将对应字符进行比较,当两个字符不同时,用前面字符串的字符的 ASCII 码值与后面字符串的对应字符的 ASCII 码值相减,于是产生大于或小于0的结果。

该函数的格式如下:

```
strcmp(s1,s2)  
strncmp(s1,s2,n)
```

其中,参数 *s1* 和 *s2* 可以是字符数组名,也可以是字符指针名。这两个函数的区别仅在于后面函数执行时,只将 *s1* 和 *s2* 的字符串中前 *n* 个字符进行比较,而其余字符不作比较。因此,后面函数中多一个 `int` 型参数 *n*。

(3) 检索字符串函数 `index()` 和 `rindex()`

该函数用来检查在指定的字符串第一次出现指定字符的位置,该函数返回一个指针,指示指定字符在字符串中出现的位置。如果字符串不包含有指定字符,则返回 `NULL`。

该函数格式如下:

```
index(s, c)  
rindex(s, c)
```

其中,参数 *s* 是表示指定字符串的字符数组名或字符指针名,*c* 是一个 `char` 变量,用来指定要检索的字符。这两个函数的区别是前一个函数是从左向右(即从头至尾)方向检索,而最后一个函数是从右向左(即从尾至头)方向检索。

(4) 字符串连接函数 `strcat()` 和 `strncat()`

该函数用来将两个已知的字符串连接起来生成一个新的字符串。该函数的返回值是一个指针,指向生成新字符串的首地址。

该函数格式如下:

```
strcat(s1, s2)  
strncat(s1, s2, n)
```

其中,*s1* 和 *s2* 是字符数组名或字符指针名,*n* 是 `int` 型数值。执行该函数则将 *s2* 字符串连接在 *s1* 字符串的尾部并去掉 *s1* 字符串的结束符 `'\0'`,连接后保留 *s2* 字符串的结束符作为连接后新字符串的结束符。返回的指针实际上是 *s1* 的首地址。使用该函数时一定要注意:*s1* 字符数组要有足够大的空间可将 *s2* 包含进去,否则越界不判错将造成数据上的混乱。这一点可通过下面的例7.17看出。函数 `strcat()` 和 `strncat()` 的区别在于前一个函数将整个 *s2* 字符串连接到 *s1* 字符串的后边,后一个函数只将 *s2* 字符串中前 *n* 个字符连接到 *s1* 字符串的后边,其余字符丢弃。

(5) 字符串复制函数 strcpy() 和 strncpy()

该函数的功能是将一个已知字符串拷贝到指定的字符数组或字符指针中。该函数返回新复制的字符串的首地址。

该函数格式如下：

```
strcpy (s1, s2)
```

```
strcpy (s1, s2, n)
```

其中,s1和s2可以是字符数组名或字符指针名,s2也可是字符常量,n是一个整型变量或常量。执行该函数则将s2中的字符串复制到s1中,s2中原有的字符串被覆盖。该函数要求s1数组的大小可以容纳下s2的所有字符,否则因越界将造成数据的混乱。strcpy()与strncpy()的区别在于前一个函数是将s2字符串全部复制到s1中,后一个函数是将s2字符串中前n个字符复制到s1中,其他字符丢弃,s2字符串保持不变,返回s1首地址。

有关字符串处理的其他函数不再一一介绍,如需要请参考所用的编译系统的说明书。

下面举些例子说明上述字符串处理函数的使用。

[例7.17] 分析下列程序的输出结果。解释出现异常情况的原因。

```
main()
{
    char *p="abcd", *q="mnopq", *strcat();
    printf("%s, %s, %s\n", p, q, strcat(p, q));
}
```

执行该程序输出结果如下：

abcdmnopq, npq, abcdmnopq

说明：出现上述输出结果是由下列原因造成的。①printf()函数参数表在使用 Turbo C 编译系统时是从右向左计算,即计算表达式 strcat(p,q),再计算 q,后计算 p。② 将 q 连到 p 后面时越界造成了数据上的混乱。请读者修改该程序使输出如下结果：

abcd, mnopq, abcdmnopq

[例7.18] 从键盘输入三个字符串,编程找出其中最小者。

假定 s1[3][10]数组用来存放用键盘上输入的3个字符串,s2[10]用来存放最小的字符串。

程序内容如下：

```
main()
{
    int i;
    char s1[3][10], s2[3][10];
    printf("Input 3 strings:\n");
    for(i=0; i<3; i++)
        gets(s1[i]);
    if (strcmp(s1[0], s1[1])>0)
        strcpy(s2, s1[1]);
    else
        strcpy(s2, s1[0]);
    if (strcmp(s2, s1[2])>0)
```

```

        strcpy(s2,s1[2]);
        printf("The least string is : %s\n",s2);
    }

```

执行该程序,显示如下提示信息:

```

Input 3 strings;
one✓
two✓
three✓
The least string is: one

```

说明:该程序中使用了有关字符串处理函数有3个:字符串输入函数 `gets()`,字符串比较函数 `strcmp()`和字符串复制函数 `strcpy()`。

[例7.19] 编写字符串连接函数 `strcat1()`。这里函数名使用 `strcat1()`是为了与系统所提供的 `strcat()`函数加以区别。

`strcat1()`函数程序如下:

```

char * strcat1(s1, s2)
register char * s1, * s2;
{
    register char * p;
    p=s1;
    while(*p++);
    p--;
    while(*p++=*s2++);
    return(s1);
}

main()
{
    char p1[15]="string", * p2="catenate";
    printf("%s\n", strcat1(p1, p2));
}

```

执行该程序输出如下结果:

```
stringcatenate
```

说明:该程序中两个函数:被调用函数 `strcat1()`是用来将两个字符串连接成一个字符串的函数,其功能与系统提供的 `strcat()`函数相同。主函数 `main()`中,先定义两个字符串,一个通过字符数组,另一个通过字符指针。然后,调用 `strcat1()`函数将已知的两个字符串连接起来并输出显示在屏幕上。

读者请思考:该程序的主函数中,将 `char p1[15]="string"`,改写为 `char * p1="string"`,是否可以?

7.3.4 指向数组的指针和指针数组

指向数组的指针和指针数组在说明时有些相似,但是这完全是两个不同的概念,前者是指针,后者是数组。

1. 指向数组的指针

指向数组的指针可以分为如下两类：

(1) 指向数组元素的指针

指向数组元素的指针是指向数组中的某个元素，可以是数组的首元素，也可以是数组的任一元素，只要将数组的某个元素的地址赋给它，则它便指向该元素。指向数组元素的指针是一级指针，它可以指向一维数组的某个元素，也可以指向二维数组或三维数组的某个元素。例如：

```
int a[10], *p;
```

```
p=&a[9];
```

p 是一个指向数组元素的指针，它指向一维数组 a 的第9个元素 a[9]。

[例7.20] 一个指向二维数组元素的指针的例子。分析下列程序输出结果。

```
int m[2][3]={ {1, 3, 5}, {7, 9, 11}};
main()
{
    int *p;
    for(p=m[0]; p<m[0]+6; p++)
    {
        if ((p-m[0])%3==0)
            printf("\n");
        printf("%4d", *p);
    }
    printf("\n");
}
```

执行该程序输出结果如下：

```
1  3  5
```

```
7  9 11
```

说明：该程序中定义了一个指针 p，它指向数组 m 的首元素，因为 m[0] 表示该数组第0行首列元素的地址，与 &m[0][0] 等价。因此，p 是一个指向二维数组 m 的首元素的指针。

请读者将 p 指针向数组 m 的尾元素，输出如下结果：

```
11  9  7
```

```
5   3  1
```

(2) 指向一维数组的指针

指向数组的指针可以指向一维数组，也可以指向二维数组，这里只讨论指向一维数组的指针。

上面讨论过指向数组元素的指针，它是指向数组的某一个元素。而这里讨论的指向一维数组的指针是指它指向的不是一个元素，而是由若干个元素组成的一维数组。例如：

```
int (*pa)[5];
```

pa 是一个指向一维数组的指针，它所指向的一维数组是由5个 int 型元素组成的。如果该指针加1，则将指向下面的5个元素。pa 指针的增值是以5个元素的一维数组长度为单位的。实际上，指向一维数组的指针，是指向二维数组的某一列的首元素地址。例如：

```
int (*pa)[5];
```

```
int a[3][5];
```

```
pa=a+1;
```

这表明 pa 是一个指向一维数组的指针,它指向二维数组 a 的第一行首列元素的地址。pa 加1将指向第二行首列元素,pa-1将指向第0行首列元素。

[例7. 21] 一个指向一维数组指针的例子。分析该程序的输出结果,说明指向一维数组的指针是怎样赋值和使用的。

```
int a[3][5]={1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
main()
{
    int (*p)[5];
    p=a+1;
    printf("%d,%d,%d\n",p[0][0],*(*(p+1)+1),*(p[-1]+3));
}
```

执行该程序输出结果如下:

6, 12, ..

说明: 程序中 p 是一个指向一维数组的指针,它所指向的一维数组有5个 int 型元素。即 p 要指向一个包含有5个元素的一维数组。p 的值是该一维数组的首元素地址。用 a+1或 &a[1] 值赋给 p,则使 p 指向 a 数组的第一行首列地址。不能用某行某列的地址给 p 赋值,

p[0][0]等价于 **p,即 a[1][0]元素的值。*(*(p+1)+1)等价于 p[1][1],即 a[2][1]元素的值。而 *(p[-1]+3)等价于 *(*(p-1)+3),即 a[0][3]元素的值。因此,输出上述结果。

实际上,指向一维数组的指针是一个二级指针,它所指向的是一个二维数组,给指向一维数组的指针所赋的值是二维数组的某行的行地址。二维数组 a[3][5]的行地址表示为 a+i,i 为0,1,2。一个指向一维数组的指针(*p)[5]可以用 a+i(i=0,1,2)来进行赋值。

请读者思考下面问题:下述的程序段是否正确。

```
int a[3][5];
int (*p)[3];
p=a;
:
```

指向数组的指针常常用来作函数的参数,这方面问题下一节讲述。

2. 指针数组

指针数组是指数组元素为指针的数组。以前讲过数组元素可以为 int 型、float 型和 double 型数值,称为数值数组;数组元素为 char 型的称为字符数组;数组元素为指针的称为指针数组,最常用的是一维一级指针数组,即该数组是一维的,它的每个元素是一个一级指针,或是一维数组,因此一维一级指针数组实际上是一个二维数组。此外,还有一维二级指针数组和二维一级指针数组,它们实际上是一个三维数组。其他的指针数组使用甚少。

指针数组在定义形式上与指向数组的指针很相似,使用中应注意区别。指针数组定义格式如下:

<类型说明符>* <数组名>[<大小>]

这是一维一级指针数组的定义格式,而指向一维数组的指针定义格式如下:

〈类型说明符〉(* 〈数组名〉) [〈大小〉]

比较可知,二者只差一个圆括号,这个圆括号的作用是改变了优先级。没有圆括号的是数组名与[]结合,则是数组,前边的*表明数组元素的类型是某种类型的指针。有了圆括号的是*与数组名结合,则是指针,后面的[]说明是指向一维数组的指针,例如:

```
int * p[5];  
int ( * g ) [5];
```

其中,p 是一维一级指针数组名,g 是指向一维数组的指针名。

一维二级指针数组和二维一级指针数组表示如下:

```
int * * a1[5], * a2[3][5];
```

其中,a1 是一维二级指针数组名,a2 是二维一级指针数组名。

指针数组的赋初值和赋值办法与数组一样,只是所赋予的值是地址值。

[例7.22] 一个指针数组的例子。分析下列程序的输出结果,说明指针数组的赋值方法。

```
int a[] = {1, 2, 3, 4, 5, 6};  
main()  
{  
    int i, * p[3];  
    for(i=0; i<3; i++)  
        p[i] = &a[i * 2];  
    printf("%d, %d\n", p[1][1], * (p[2] + 1));  
}
```

执行该程序输出如下结果:

4,6

说明:该程序中 p 是一个一维一级指针数组名,它有3个元素,即每个元素是指向 int 型的指针。通过 for 循环给 p 赋值,分别使 p[0] 值为 &a[0],p[1] 值为 &a[2],p[2] 值为 &a[4]。

指针数组比较适合于对若干字符串的处理。字符串本身是一个一维字符数组,如将若干个字符串放到一个二维字符数组中,则每一行的元素个数要求相同,实际上各字符串长度不等,只好按字符串中最长的作为每行的元素个数。这样将会造成内存空间的浪费,如果采用指针数组便可克服存储空间的浪费问题。因为指针数组中各个指针元素可以指向不同长度的字符串。因此,实际编程中常用字符型的指针数组存放字符串。

[例7.23] 分析下列程序的输出结果。

```
char * name[] = {"", "Monday", "Tuesday", "Wednesday", "Thursday",  
                "Friday", "Saturday", "Sunday"};  
main()  
{  
    int week;  
    while(1)  
    {  
        printf("Enter week No. : ");  
        scanf("%d", &week);  
        if (week < 1 || week > 7)  
            break;  
        printf("week No. %d --> %s\n", week, name[week]);  
    }
```

```

    }
}

```

执行该程序输出如下结果：

```

Enter week NO. :3✓
week NO. 3 -->Wednesday
Enter week NO. :7✓
Week NO. 7 -->Sunday
Enter week NO. :0✓

```

退出该程序。

说明：该程序中 name 是一个字符型的指针数组名，用它来存放字符串。使用字符型指针数组存放不等长度的字符串不浪费存储空间。该程序的功能是将输入的数字表示的星期几转换为英文单词表示的星期几。使用指针数组将数组下标的数字与英文单词对应起来，十分方便的完成该程序的功能。

[例7.24] 一个二维一级字符指针的例子。

分析下列程序的输出结果：

```

char *p[2][3]={"abc","def","ghijk","lmno",
               "pqrstuvw","xyz"};

main()
{
    printf("%c\n", p[1][2][2]);
    printf("%c\n", * * * p);
    printf("%c\n", * * p[1]);
    printf("%c\n", * * (p[1]+2));
    printf("%c\n", ( * ( * (p+1)+1))[2]);
    printf("%c\n", * (p[1][1]+4));
    printf("%c\n", ( * (p[0]+1))[2]);
    printf("%c\n", * ( * ( * (p+1)+2)+1)+1);
}

```

执行该程序输出的如下结果：

```

z
a
l
x
r
t
f
z

```

说明：该程序中定义了一个字符型的二维一级指针数组 p，它有6个元素，每个元素是一个指向 char 型的一级指针，即指向一个字符串，该数组共有6个字符串，它们的长度不同，用指针数组表示比较方便。

二维一级指针数组实际上是一个三维数组，它的每个元素是一个字符。该程序中将 p 看成一个三维的 char 型数组名，使用三维数组元素的各种不同表示方法来输出数组元素的字

符。前面讲过了三维数组各元素的不同表示法。经过分析知：

`***p` 等价于 `p[0][0][0]`，即首元素 `a`。

`**p[1]` 等价于 `p[1][0][0]`，由于 `p` 数组将6个字符串分成2行，每行3个。`p[1][0][0]` 是指第1行第0列字符串中的首元素，即 `l`。

`*(p[1]+2)` 等价于 `p[1][2][0]`，即第1行第2列的字符串中首元素，即 `x`。

`((*(p+1)+1))[2]` 等价于 `p[1][1][2]`，即第1行第1列的字符串中第2个字符，即 `r`。

`*(p[1][1]+4)` 等价于 `p[1][1][4]`，即第1行第1列的字符串中第4个字符，即 `t`。

`((*(p[0]+1))[2]` 等价于 `p[0][1][2]`，即第0行第1列的字符串中第2个字符，即 `f`。

`((*(p+1)+2)+1)+1` 等价于 `p[1][2][1]+1`，而 `p[1][2][1]` 是第1行第2列字符串中第1个字符，即 `y`，`'y'+1` 则为字符 `z`。

指针数组可以看成是多维数组，也可以看成是多级指针。例如，一维一级指针组可以认为是二维数组，因为一级指针可看成是一维数组；也可以认为是二级指针，因为一维数组可看为一级指针，于是一维一级指针可看成是指针的指针，即二级指针。

在C语言中，数组和指针是可以相互转换的，多维数组可以转换为多级指针。例如，一维数可用一级指针表示，二维数组可用二级指针表示，三维数组可用三级指针表示等等。

[例7.25] 使用图解法分析下列程序的输出结果。

程序内容如下：

```
char *str[3][2]={"abc", "def", "ghijk", "lmno",
                "pqrstuvw", "xyz"};
char **pc[]={str[2], str[0], str[1]};
char ***ppc=pc;
main()
{
    printf("%s\n", *ppc[2]);
    printf("%s\n", **ppc+2);
    printf("%s\n", *++*++ppc+2);
    printf("%s\n", ppc[0][-1]+1);
    printf("%s\n", *((*ppc-1)+1));
}
```

执行该程序输出结果如下：

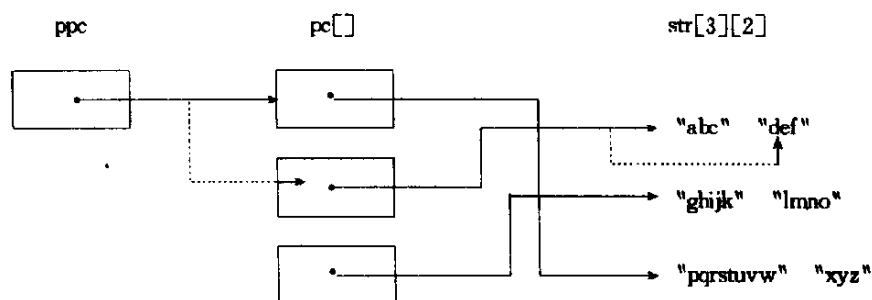
```
ghijk
rstuvw
f
bc
xyz
```

说明：

(1) 图解法是一种对程序的辅助分析方法。有些程序中变量之间关系比较复杂，为了理顺关系可以使用图示的方法直观清楚地将各变量之间关系表示出来，在使用指针数组的程序中常用这种方法。下面以本题为例说明图解法的用法。

本题中有三个变量：`str` 是一个二维一级指针数组，它存放了6个不同长度的字符串。`pc` 是一个一维二级字符数组，它有3个元素，每个元素是一个二维字符数组或一维一级字符指针数

组。ppc 是一个三级字符指针。根据程序中给出关系,将它们用图解表示如下:



从上图中可以看出 ppc 是一个指向 pc 数组首元素的指针。pc 数组有3个元素,每个元素是一个指针,pc[0]指向 str[2],pc[1]指向 str[0],pc[2]指向 str[1]。

具体分析: ppc 是一个三级指针,而 pc 可认为是三维数组的数组名,或认为是一个三级指针名。pc 有3个元素,每个元素应该是一个二级指针。str 是一个二维一级指针数组名, str[i] 也是一个数组名,它是一个一维一级指针数组名,它有2个元素,每个元素是一个字符串。

(2) 进一步分析5个 printf() 语句的输出结果。

第一个 printf() 语句,按 %s 输出 * ppc[2] 的值,由于 ppc 指向 pc, ppc[2] 等价于 * (ppc + 2),即指向 str[1], * ppc[2] 等价于 * (ppc[2] + 0),即是字符串 "ghijk" 首元素地址。因此,输出字符串为 ghijk。

第二个 printf() 语句,按 %s 输出 * * ppc + 2 的值。* * ppc 等价于 * (* (ppc + 0) + 0),即是 str[2] 数组的首列元素的首地址值,即字符串 "pqrstuvw" 的首元素地址值, * * ppc + 2 则是该字符串中的 r 字符的地址值。因此,输出字符串为 rstuvw。

第三个 printf() 语句,按 %s 输出地址值 * + + * + + ppc + 1 所指的字符串。* + + ppc 等价于 pc[1],它指向 str[0],而 * + + * + + ppc 将是指向字符 "def" 的指针, * + + * + + ppc + 2 是字符串 "def" 中字符 f 的地址。因此,输出字符串为 f。这里需要注意的是 ppc 由于经过 ++ 运算后,已经指向 pc[1] 了,同理, str[0] 也已经指向字符串 "def" 了。这便是由 ++ 运算符副作用所造成的。如图中虚线所示。

第四个 printf() 语句,按 %s 输出 ppc[0][- 1] + 1 地址值所指向的字符串。ppc[0][- 1] 等价于 * (* (ppc + 0) - 1),这时 ppc 指向 pc[1], ppc[0][- 1] 指向字符串 "abc" 首地址(因为 ppc[0][0] 已是指向 "def" 的首地址), ppc[0][- 1] + 1 指向字符串 "abc" 中的 b 字符。因此,输出字符串为 bc。

第五个 printf() 语句,将 %s 输出 * (* (ppc - 1) + 1) 所指向的字符串。* (* (ppc - 1) + 1) 等价于 ppc[- 1][1],这时 ppc 仍指向 pc[1], * (ppc - 1) 指向 pc[0], * (* (ppc - 1) + 1) 指向字符串 "xyz" 的首地址。因此,输出字符串为 xyz。

7.4 指针与函数

这一节中要讲解指针在函数中的应用。指针作函数参数可以实现传址调用。指针作为函数的返回值,该函数称为指针函数。指向函数的指针可以作为函数参数,实现特殊的函数调用。指

针在函数中有着广泛的应用,在 C 语言的程序中应该很好地发挥指针的作用。

7.4.1 指针作为函数的参数

指针作为函数的参数前面已经讲过了。使指针作函数的形参时,要求实参用地址值,实现变量地址的传递,这种调用称为传址调用。它的特点是在被调用函数中通过改变形参的内容(即形参所指的变量的值)来改变调用函数中实参的值。

1. 基本数据类型的指针作函数参数

关于基本数据类型的指针作函数参数前面已经讲过了,并在“函数和存储类”一章中举过了一些例子。这里不再重复。

指针作为函数参数可以实现函数之间的数据传递。即可将被调用函数中的值通过参数传递给调用函数,这种传递既安全可靠,又可一次实现多个数据传递,这是常用的传递方式。

2. 数组名作函数参数

C 语言中,数组名是一个指针,是首元素的地址值。用数组名作函数参数,实现的是传址调用。数组名可以作函数的形参,也可以作函数的实参,另外,可用数组名作实参,用指针作形参,还可用数组名作形参,指针作实参。这些都是属传递地址值的传址调用。

【例7.26】采用选择法对数组中的若干整数按由小到大进行排序。选择法排序做法如下:先从所有的整数中找出最小的作为首元素,再从剩下整数找出最小的作为首元素后边的元素,依次类推,每次从未排序的整数中找出最小的一个。如果有 n 个整数,需要这样比较查找 $n-1$ 次。程序内容如下:

```
int a[8]={5,9,10,18};
main()
{
    int i, *pa;
    pa=a;
    sort(pa,8);
    pa=a;
    for(i=0;i<8;i++)
        printf("%4d", *pa++);
    printf("\n");
}
sort(b,n)
int b[],n;
{
    int i,j,k,t;
    for(i=0;i<n-1;i++)
    {
        k=i;
        for(j=i+1;j<n;j++)
            if(b[j]<b[k])
                k=j;
        if(k!=i)
        {
            t=b[i];
            b[i]=b[k];
            b[k]=t;
        }
    }
}
```

```

        b[i]=b[k];
        b[k]=t;
    }
}
}

```

执行该程序输出结果如下:

```
3 5 6 9 10 13 18 21
```

说明: 该程序中有两个函数: `main()` 和 `sort()`。在 `main()` 函数中, 定义指针 `pa`, 它指向数组 `a` 的首元素。调用 `sort()` 函数, 使用指针 `pa` 作实参, 另一个实参是数值 8。在 `sort()` 函数中有二个形参, 一个是数组名 `b`, 另一个是 `int` 型变量 `n`。调用时, 实参 `pa` 将它所指的数组的首元素地址传递给数组 `b`, 这样使得实参所指的数组 `a` 和形参数组 `b` 共用一段内存空间。即两个数组的同一个元素共占一个存储单元。因此, 在 `sort()` 函数中直接改变了外部数组 `a` 的元素值, 即在被调用函数中实现对数组 `a` 的排序。程序最后通过 `for` 循环将排序后的数据输出显示。这里, `printf()` 函数使用了 `*pa++` 输出, 在循环输出之前, 使用了赋值表达式语句 `pa=a`; 请读者思考, 不用这条语句行吗? 为什么?

读者还可以在该程序的基础上对调用函数和被调用函数的参数进行如下修改:

- ① 形参实参都用数组;
- ② 形参实参都用指针;
- ③ 形参用指针实参用数组。

看一下输出结果是否有变化。

3. 指向数组的指针作函数参数

前面讲过了指向一维数组元素的指针作函数的例子。这里着重讲解指向一维数组的指针作函数参数。

[例7.27] 假定有5个学生, 每个有3门功课的成绩。编程输出某个学生的成绩。
编程如下:

```

float score[5][3]={{86, 90, 78}, {86, 84, 96}, {91, 76, 88},
                  {90, 77, 66}, {78, 97, 68}};

main()
{
    int n;
    printf("Input student No. (1--5):\n");
    scanf("%d", &n);
    search(score, n);
}

search(p,a)
float (*p)[3];
int a;
{
    int i;
    printf("the scores of student No. %d are:\n", a);
    for(i=0; i<3; i++)
        printf("%5.1f", *(*(p+a-1)+i));
    printf("\n");
}

```

```
}
```

执行该程序显示如下信息:

```
Input student No. (1-5):  
2  
the scores of student No. 2 are:  
86.0 84.0 96.0
```

说明: 该程序中 `search()` 函数的一个形参为指向一维数组的指针 `p`。调用时, 对应的实参是一个数组名 `score`, 它是一个二维数组。一般地讲, 一个指向一维数组的指针作为形参时, 调用时所对应的实参为同类型的二维数组的名字。由于数组下标是从 0 开始而学生的号码是从 1 开始, 因此, 在输出成绩时作了一点变化。

请读者考虑一下, 将函数 `search()` 的第一个形参改为二维数组是否可以? 即将

```
float (*p)[3];
```

改为

```
float p[5][3];
```

4. 字符指针作函数参数

字符指针是指向字符串的指针。该指针实际上是指向一维字符数组首元素的指针, 即指向字符串的首字符的指针。

[例 7.28] 编程通过调用自己编写的字符串拷贝函数来实现字符串的复制。

编程内容如下:

```
main()  
{  
    char s1[20]="I love China."  
    char s2[]="I love Beijing."  
    printf("s1: %s\ns2: %s\n", s1, s2);  
    strcpy(s1, s2);  
    printf("s1: %s\ns2: %s\n", s1, s2);  
}  
strcpy(p, q)  
char *p, *q;  
{  
    while((*p++ == *q++) != '\0')  
        ;  
}
```

执行该程序输出如下结果:

```
s1: I love China.  
s2: I love Beijing.  
s1: I love Beijing.  
s2: I love Beijing.
```

说明: 字符串拷贝函数 `strcpy()` 中的两个形参都是指向字符的指针。调用时对应的实参是字符数组名, 也可是字符指针, 读者可改写成字符指针作实参试一试。

在 `strcpy()` 函数中, `while` 循环可以写成如下更加简练的形式:

```
while (*p++ == *q++);
```

这样写法在 Turbo C 编译系统下会出现什么问题呢?请读者思考。

5. 指向指针的指针作函数参数

指向指针的指针是一个二级指针,该指针所指向的变量还是一个指针,而这个指针指的是一个非指针变量,如果这个指针所指的仍然是一个指针,则该指针为三级指针。

[例7.29] 编程实现对若干个字符串进行排序,并由小到大输出。

编程内容如下:

```
char * name[] = {"FORTRAN", "C", "BASIC", "PASCAL", "LISP"};
main()
{
    int n=5;
    strsort(name,n);
    printstr(name,n);
}
strsort(p,n)
char ** p;
int n;
{
    int i,j,k;
    char * x;
    k=n/2;
    while(k>=1)
    {
        for(i=k;i<n;i++)
        {
            x=p[i];
            j=i-k;
            while(j>=0&&strcmp(x,p[j])<0)
            {
                p[j+k]=p[j];
                j-=k;
            }
            p[j+k]=x;
        }
        k/=2;
    }
    printstr(q,n)
    char ** q;
    int n;
    {
        int i;
        for(i=0;i<n;i++)
            printf("%s\n", *(q+i));
    }
}
```

执行该程序输出如下结果:

BASIC
C
FORTRAN
LISP
PASCAL

说明:

(1) 该程序由一个主函数 `main()` 和二个被调用函数 `strsort()` 和 `printstr()` 组成。其中, `strsort()` 函数是一个字符串排序函数, 它有一个二级指针的参数, 调用时它将指向一维一级指针数组 `name`。 `printstr()` 函数是一个打印字符串的函数, 该函数也有一个二级指针的形参, 调用时它也指向指针数组 `name`。这两个函数都有指向指针的指针作函数的形参。调用时, 该参数对应实参应是一个二维数组名, 或一维一级指针数组名, 二级指针名也可以。

(2) 字符串排序函数 `strsort()` 中采取了 shell 排序法。这是一种效率较高的排序方法之一, 这种排序法又称缩小增量法。具体操作如下: 在 n 个数据项组成的数据序列中, 首先取一个整数 $d_1 < n$, 把全部数据项分成 d_1 组, 所有相隔距离为 d_1 倍数的数据项放在一组中, 在各组内进行排序; 然后, 取 $d_2 < d_1$ 重复上述分组和排序工作; 直到 $d_i = 1$ 即所有的都放在一个组中排序为止。各组内排序可采用直接插入法。一般地, 开始时 d 取值较大, 逐渐减小。此例中, $n = 5$, 则增量取值为 2, 1。因为采取了 $n/2$ 取整的方法。

分组后各组排序采用的是直接插入法。这又是排序的一种方法。直接插入法具体操作如下: 该排序方法是首先对数据序列的前两个数据进行排序, 然后根据前两个数据排序的情况将第 3 个数据插入到适当位置。再根据前 3 个数据排序的情况将第 4 个数据插入到适当位置, 依此类推直到所有数据排好为止。

6. 指针数组作函数参数

指针数组作函数参数实际上与数组名作函数参数和指向指针的指针作函数参数是相同的。在前面讨论的例 7.29 中, 两个被调用函数 `strsort()` 和 `printstr()` 中第一个参数可以改写为用指针数组的形式表示。即将

```
char **p;
```

改写为

```
char *p[];
```

请读者上机试一试这样更改是否可以。

指针数组可以作主函数 `main()` 的参数, 即用该指针数组来存放命令行参数。以前讲过的主函数 `main()` 都是不带参数的, 这样的程序经编译连接成可执行的文件(或命令), 在运行这种命令时, 一般不得带有参数。如果希望在执行命令时通过指定某些参数, 并将它们传递给所执行的程序, 则需要该程序的主函数 `main()` 带有参数。这就是说, 只有程序的主函数带有参数才允许执行该程序的命令行带参数。

主函数 `main()` 带有参数的格式如下所示:

```
main(argc, argv)
int argc;
char *argv[];
{
```

...

}

主函数一般带有2个参数,一个是 int 型变量 argc,它用来存放执行该程序时命令行参数的个数,实际上是参数个数加1,这包含了命令字。另一个参数是 char 型一维一级指针数组,它用来存放命令行参数的字符串的,包含命令字在内。并规定,argv[0]存放命令字的字符串的,argv[1]存放第1个参数(命令字后面首参数),argv[2]存放第2个参数,以此类推,argv[]数组的大小,由 argc 的值来确定。

[例7.30] 编程验证主函数参数 argv 是用来存放命令行参数的字符串的。

程序内容如下:

```
main(argc,argv)
int argc;
char *argv[];
{
    int i;
    for(i=1;i<argc;i++)
        printf("%s%c",argv[i],(i<argc-1)?' ':'\n');
}
```

将该程序编译连接后生成的可执行文件名为 clp.exe。假定命令行如下:

clp prog1.c prog2.c prog3.c ✓

执行结果如下:

prog1.c | prog2.c | prog3.c

说明:该程序结果告诉我们,主函数参数 argv[]是用来存放命令行参数字符串的,该例中,argv[1]存放"prog1.c",argv[2]存放"prog2.c",argv[3]存放"prog3.c"。

读者可以试一试,argv[0]中是否存放的"clp"呢?

7.4.2 指针函数和指向函数的指针

1. 指针函数

指针函数是指返回值是指针的那类函数。不同类型的指针可以作为函数的返回值。前面讲过的系统提供的一些字符串处理函数中有些就是返回值为字符指针(char *)的函数。例如:

char *strcat (s1, s2)

char *strcpy (s1, s2)

指针函数的定义格式如下:

〈类型说明符〉*〈函数名〉(〈参数表〉)

例如,

int *fun(x, y)

这是一个指针函数,函数名是 fun,该函数有二个参数 x 和 y,该函数的返回值是一个 int 型指针。

[例7.31] 某车间有4个班组,已知去年每个班组每个季度的产值(单位:万元)。编程实现在输入某个班组序号后,输出该班组的每季度产值和全年总产值。要求用指针函数实现。

编程内容如下:


```

float out[][4]={{4.5, 5.2, 4.3, 5.0}, {5.7, 6.1, 4.9, 5.3},
               {4.9, 5.1, 4.8, 4.2}, {6.0, 5.5, 5.7, 6.1}};

main()
{
    float *pf, *search(), s=0.0;
    int i, n;
    printf("Input teams and groups(1---4): \n");
    scanf("%d", &n);
    printf("The output value of teams and groups No. %d:\n", n);
    pf=search(out, n);
    for(i=0; i<4; i++)
    {
        s += *(pf+i);
        printf(" %6.2f", *(pf+i));
    }
    printf("\ts= %6.2f\n", s);
}

float *search(out1, n)
float (*out1)[4];
int n;
{
    float *pf1;
    pf1 = *(out1+n-1);
    return(pf1);
}

```

执行该程序,显示如下提示信息:

Input teams and groups (1---4): 2✓

输出信息如下:

The output value of teams and groups No. 2
5.70 6.10 4.90 5.30 s=22.00

说明:该程序中 search()函数是一个指针函数,其返回值为(float *),即指向浮点型变量的指针,该指针将指向 float 型数组 out 的某一行的首列,即返回值 pf1 为指向 out1 数组某一行首列地址,将它赋值给 pf,而 *(pf+i)便是该行的各个元素的值。

请读者思考,如在 search()函数中,将作如下改动,会发生什么变什呢?

pf1 = *(out1+n-1);

改为

pf1 = *out1+n-1;

请读者自己分析。

[例7.32] 编程输出上例中年产值不足20万元的班组号及年产值。

编程内容如下:

```

float out[][4]={{4.5,5.2,4.3,5.0},{5.7,6.1,4.9,5.3},
               {4.9,5.1,4.8,4.2},{6.0,6.5,5.7,6.1}};

```

```

main()
{
    float * pf, * search(), s;
    int i, j;
    for(i=0; i<4; i++)
    {
        s=0.0;
        pf=search(out+i);
        if(pf==*(out+i))
        {
            printf("No. %d year outpt value: ", i+1);
            for(j=0; j<4; j++)
                s+=*(pf+j);
            printf(" %6.2f\n", s);
        }
    }
}

float * search(out1)
float (*out1)[4];
{
    int i;
    float * pf1, s=0.0;
    for(i=0; i<4; i++)
        s+=*(*out1+i);
    if(s<20.0)
        pf1=*out1;
    else
        pf1=0;
    return(pf1);
}

```

执行该程序输出如下结果：

No. 1 year output value: 19.00

No. 3 year output value: 19.00

说明：该程序中 search() 函数是一个返回值为指向 float 型的指针，并且该函数的形参是一个指向数组的指针。该函数返回的指针是一个一级指针，它指向数组的某一行的首元素。

2. 指向函数的指针

指向函数的指针的定义格式如下：

〈类型说明符〉(* 〈指针名〉) () ;

例如，

```
int (* pf)();
```

其中，pf 是一种指向函数的指针名。该指针所指向的函数的返回值是一个 int 型数。

指向函数的指针要用一个函数名来给它赋值，用哪个函数给它赋值，它就指向那个函数。

例如，

sin(x) 是一个系统提供的放在 math.h 文件的一个数学函数。定义一个指向函数的指针，

```
double (* p)();
```

```
p=sin;
```

则 p 指针指向 sin()函数。

用一个函数名给指向函数的指针赋值,则该指针指向该函数所存放在内存中的入口地址。这就是说,一个函数存放在内存中的入口地址是用函数名来表示的,而不必给出参数,也不要圆括号。

用指向函数的指针调用函数时,只需使用如下格式:

```
(*(指针名))(<实参表>)
```

例如,

```
int (*p)(), add();
```

```
p=add;
```

```
z=(*p)(x, y);
```

其中,函数 add()是带有2个形参的用来求两个数的和的函数。通过(*p)(x, y)来调用,求 x 和 y 的和,赋值给 z。

[例7.33] 使用指向函数的指针调用函数。

```
main()
{
    int add(), (*p)();
    int x, y, z;
    p=add;
    printf("Input x, y: ");
    scanf("%d%d", &x, &y);
    z=(*p)(x, y);
    printf("x=%d, y=%d, sum=%d\n", x, y, z);
}

int add(a, b)
int a, b;
{
    return(a+b);
}
```

执行该程序,显示如下提示信息:

Input x, y: 18 27 ↵

输出如下:

x=18, y=27, sum=45

说明:

(1) 程序中, p 是一个指向函数的指针。

```
p=add;
```

这是用函数名给指向函数的指针赋值。

(2) 程序中,

```
z=(*p)(x, y);
```

是用指向函数的指针来调用该指针所指向的函数,它等价于

```
z=add(x, y);
```

(3) 需要说明的一点是：该程序中主函数内使用 `int add()`；对函数进行说明这是必要的，否则编译将会出错。前面曾经说过返回值为 `int` 型的函数在调用前可以不去说明，那是指函数调用的情况下，不必去说明。可是，这里是将一个函数名赋给一个指针，使用时没有加圆括号和参数，编译系统难以判断是变量名还是函数名，因此，对这种只使用函数名的情况下，对该函数需要说明，这一点使用时应该特别注意。

指向函数的指针通常用作函数的参数。当函数的形参用指向函数的指针时，该函数在调用时，实参要用函数名，即将某个函数在内存中的入口地址赋给对应的指向函数的指针，实现函数地址的传递。

[例7.34] 指向函数的指针作函数参数。

分析下列程序的输出结果：

```
#include <math.h>
main()
{
    double sin(), cos(), tan();
    double x;
    int a;
    printf("Input a : ");
    scanf("%d", &a);
    x = 3.1415/180 * a;
    printf("sin(%d)=", a);
    fun(x, sin);
    printf("cos(%d)=", a);
    fun(x, cos);
    printf("tan(%d)=", a);
    fun(x, tan);
}
fun(y, f)
double y, (*f)();
{
    printf("%.2lf\n", (*f)(y));
}
```

执行该程序，显示如下提示信息：

Input a: 45 ✓

输出如下结果：

sin(45)=0.71

cos(45)=0.71

tan(45)=1.00

说明：该程序是使用指向函数的指针作函数参数的一个例子。程序中 `fun()` 函数有一个形参是用指向函数的指针 `f`，调用时调用函数对应的实参是函数名。该例中 `main()` 内三次调用 `fun()` 函数，每次使用的函数名分别是 `sin`，`cos`，`tan`。另外，该程序中是通过函数调用求 `sin(x)`，`cos(x)` 和 `tan(x)` 的值，三个数学函数要求是弧度值，而从键盘上输入的是角度值，如45表示45度角，因此，程序中有个转换的表达式。

练习题

1. 什么是指针?为什么说指针是一种特殊变量?它与一般变量有何不同?
2. 如何表示变量的地址值?运算符 & 在使用时应注意些什么问题?
3. 各种不同类型的指针都如何表示?指向数组的指针和指针数组在表示上有何区别?指向函数的指针和指针函数在表示上有何区别?
4. 定义一个指针后不赋值或不赋初值能否使用?举例说明不同类型的指针如何赋值?
5. 指针赋值时应注意些什么问题?给一个指针赋值为 NULL 是什么意思?
6. 如何确定某个指针所指向的变量?单目运算符 * 在使用时应注意些什么问题?
7. 指针具有哪些运算?为什么说“指针运算实际上是地址运算,但又与地址运算不同”?
8. 如何判断是指针运算还是地址运算?
9. 两个指针可以相减,也可以进行比较运算,这两种指针运算是否是无条件的?为什么?
10. 在下列的定义中, p 和 a 都是指针,它们有何区别?

```
int a[10], *p=a;
```
11. 指向数组的指针和指向数组元素的指针有何区别?
12. 写出一维数组、二维数组和三维数组的下标、指针的各种表示形式。
13. 什么是字符数组?什么是字符指针?字符指针的引进有什么好处?
14. 有哪些常用的字符串处理函数?它们的功能是什么?
15. 指向数组的指针和指针数组都有哪些用途?
16. 指向函数的指针作函数的参数和指针数组作函数的参数对其实参有何要求?
17. 什么是指针函数?指针函数和指向函数的指针都有哪些用途?
18. 指针作函数形参与一般变量作函数形参有何不同?
19. 指向函数的指针作函数形参时对应的实参应该是什么?
20. 总结下述问题:
 - (1) 指针到底是什么?
 - (2) 使用指针时应该注意哪些问题?
 - (3) 在实际应用中,指针会带来什么好处?

作业题

1. 判断下列描述是否正确,对者划√,错者划×。
 - (1) 指针是用来存放某个变量的地址值的变量,那么指针的类型是地址值的类型,即为 unsigned int。
 - (2) 在 C 语言中,任何一个数组名都是一个常量指针。
 - (3) 如果 p 是一个指向 int 型变量的指针,则表达式 *p 表示 p 所指向的 int 型变量的值。
 - (4) 一个指针只能指向与它类型相同的变量。
 - (5) 一维字符数组就是字符串。
 - (6) 给指针赋值只要赋一个地址值就可以了。
 - (7) 给一个指针赋值 NULL(实际上 NULL 为 0)与不给它赋值是等价的。
 - (8) 任何两个毫无相关的指针作相减运算是没有意义的。
 - (9) 字符数组不可以直接赋值一个字符串,而字符指针可以直接赋值一个字符串。
 - (10) 在使用数组名作实参和形参的函数调用时,实参数组和形参数组共用一段内存单元的。

(11) C 语言中,数组元素可以用下标表示,也可以用指针表示。用指针表示比用下标表示效率更高。

(12) 指向函数的指针应该赋给一个所指向的函数的入口地址值,该地址值表示如下:

&((函数名))

(13) 在一个二维数组 $a[3][5]$ 中, $a[0]$ 与 $*a$ 是等价的, $\&a[1]$ 与 $a+1$ 也是等价的。

(14) 任何类型的指针没有赋值或赋初值就使用是危险的。

(15) 给一个指向 int 型变量的指针赋值,只有将该变量的地址值赋给它,再无其他办法。

2. 选择填空

(1) 指向同一个数组的两个指针,作()运算是没有意义的。

A. 相加 B. 相减 C. 比较 D. 赋值

(2) 已知, $\text{int } a, *pa=\&a$; 则 $\&*pa$ 与()是相同的。

A. pa B. a C. $\&a$ D. $\&pa$

(3) 已知, $\text{int } a[5], *p=a$; 则 $++*p$ 与()是相同的。

A. $*++p$ B. $*++a$ C. $*p++$ D. $++a$

(4) 已知, $\text{int } m, *p=\&m$; 则 $*\&m$ 与()是相同的。

A. m B. $*m$ C. p D. $*p$

(5) 已知, $\text{int } a, *pa; \text{char } c, *pc$; 下列()是合法的。

A. $pa=\&c$ B. $pa=a$ C. $pa=pc$ D. $pa=(\text{int } *)\&c$

(6) 已知, $\text{int } i=5, *p=\&i$; 下列表达式中()是非法的。

A. $i=20$ B. $*p=10$ C. $*\&i=15$ D. $p=i$

(7) 已知, $\text{int } a[]=\{5, 4, 3, 2, 1\}, *p=a$; 下列对数组元素的引用中()是非法的。

A. $*(a+2)$ B. $a(p-a)$ C. $*\&a[1]$ D. $++p$

(8) 已知, $\text{int } a[]=\{1, 2, 3, 4, 5\}, *p=a$; 下列对数组元素地址的引用中()是正确的。

A. $\&(a+1)$ B. $\&(p+1)$ C. $\&p[2]$ D. $p++$

(9) 下列对于 $\text{int } *p[5]$; 中标识符 p 的说法,()是正确的。

A. p 是一个指向一维数组的指针
B. p 是一个一维一级指针数组,有5个元素,每个元素是一个指向 int 型的指针
C. p 是一个指向 int 型变量的指针
D. p 是一种数组指针

(10) 已知, $\text{int } a[3][5]$; 下列表示数组 a 的元素中,()是错误的。

A. $** (a+1)$ B. $(*(a+1))[2]$ C. $*(*(a+1)+2)$ D. $*(a+2)$

(11). 下列给数组赋初值的表示中,()是错误的。

A. $\text{int } a[][3]=\{\{1, 2\}, \{3\}, \{4, 5\}\};$

B. $\text{int } a[2][4]=\{1, 2, 3, 4, 5\};$

C. $\text{int } a[3][4]=\{1, 2\};$

D. $\text{int } a[4]=\{1, 2, 3, 4, 5\};$

(12) 已知, $\text{int } a[3][4], (*p)[4]$; 下列赋值表达式语句中,()是正确的。

A. $p=a+2;$ B. $p=a[2];$ C. $p=*a;$ D. $p=*a+2;$

(13) 已知, $\text{int } a[3][4], *p, *q[3]$; 下列赋值表达式语句中,()是正确的。

A. $p=a;$ B. $p=*a;$ C. $q=a+1;$ D. $q=\&a[1][2];$

(14) 已知, $\text{int } a[3][4], (*p)[4], p=a$; 表示数组元素 $a[2][0]$ 的地址是()。

A. $** (a+2)$ B. $*p+2$ C. $\&a[2]$ D. $*(p+2)$

(15) 已知, $\text{int } b[3][2], (*q)[2]; q=b$; 表示数组元素 $b[1][0]$ 的值是()。

A. $*(*b+1)$ B. $*(q+1)$ C. $*(b+1)[0]$ D. $** (q+1)$

(16) 在某程序的主函数中使用了如下的调用函数语句,

```
a=fun (f1);
```

其中,f1是一个已知函数的函数名,该函数有一个参数.fun()函数定义如下:

```
int fun (x1)
int (* x1) ();
{
    ...
}
```

在 fun()函数中,返回调用 f1()函数的返回值语句是()。

A. return ((* x1)(a));

B. return (* x1(a));

C. return (x1(a));

D. return (&x1(a));

3. 分析下列各程序的输出结果

(1)

```
main()
{
    static int x[]={1, 2, 3, 4};
    int *p, i, a, b;
    p=&x[1];
    a=10;
    for(i=3; i>=0;i--)
        b=( * (p+i)<a)? * (p+i); a;
    printf("%d\n", b);
}
```

(2)

```
main()
{
    int a, b, c;
    fun(1, 2, &a);
    fun(3, a, &b);
    fun(a, b, &c);
    printf("%d, %d, %d\n", a, b, c);
}

fun(i, j, k)
int i, j, *k;
{
    j*=i;
    *k=i+j;
}
```

(3)

```
int a[]={9,7,5,3,1};
main()
{
```

```

    int i, *p=a;
    fl(p, 0, 1);
    fl(p, 1, 2);
    fl(p, 2, 3);
    fl(p, 3, 4);
    for(i=0; i<5; i++)
        printf("%5d", *(a+i));
    printf("\n");
}
fl(x, y, z)
int *x, y, z;
{
    int c;
    while(y<z)
    {
        c = * (x+y);
        * (x+y) = * (x+z);
        * (x+z) = c;
        y++;
        z--;
    }
}

```

(4)

```

#define PR(x) printf("%5d",x)
int a[]={8, 7, 6, 5, 4, 3, 2, 1}, *p;
main()
{
    int i;
    p=a+2;
    for(i=3; i; i--)
    {
        switch(i)
        {
            case 1:
            case 2: PR(*++p);
                    break;
            case 3: PR(*--p);
        }
    }
    printf("\n");
}

```

(5)

```

main()
{
    int n;
    char *p1, *p2;

```



```

    p1="abcxyz";
    p2="abcijk";
    n=fun(p1, p2);
    printf("%d\n", n);
}
fun(s1, s2)
char *s1, *s2;
{
    while( *s1&&*s2&&*s2++==*s1++);
    return(*s1-*s2);
}

```

(6)

```

main()
{
    static int a[10]={12, 10, 9, 6, 5, 4, 2, 1}, i, n, x;
    n=i=8;
    x=3;
    while(x>* (a+i))
    {
        * (a+i+1)=* (a+i);
        i--;
    }
    * (a+i+1)=x;
    for(i=0; i<=n+1; i++)
        printf("%5d", * (a+i));
    printf("\n");
}

```

(7)

```

main()
{
    int a[3][5], i, j;
    for(i=0; i<3; i++)
        for(j=0; j<5; j++)
            a[i][j]=i*5+j;
    for(i=0; printf("\n"), i<3; i++)
        for(j=0; j<5; j++)
            printf("%8d", * (* (a+i)+j));
    printf ("\n%8d%8d%8d%8d%8d%8d\n", * *a, * * (a+1),
        * (a[0]+2), * (*a+2), (* (a+1))[2], * (* (a+2)+3));
}

```

(8)

```

int a[3][3]={1,2,3,4,5,6,7,8,9};
int *p[]={a[0],a[1],a[2]};
int **pp=p;
main()

```

```

{
    int (*s)[3]=a, *q=&a[0][0];
    int i,j;
    for(i=1;i<3;i++)
        for(j=0;j<2;j++)
        {
            printf("%d,%d,%d\n",*(a[i]+j),
                *((p+i)+j),*((pp+i))[j]);
            printf("%d,%d\n",*(q+3*i+j),*(s+3*i+j));
        }
}

```

(9)

```

int a[5][5]={0};
main()
{
    int i,j,*p[5];
    for(i=0;i<5;i++)
        *(p+i)=&a[i][0];
    for(i=0;i<5;i++)
    {
        *(p[i]+i)=1;
        *(p[i]+5-i-1)=1;
    }
    for(i=0;i<5;i++)
    {
        for(j=0;j<5;j++)
            printf("%4d",p[i][j]);
        printf("\n");
    }
}

```

(10)

```

char *c[]={"ENTER","NEW","POINT","FIRST"};
char **cp[]={c+3,c+2,c+1,c};
char ***cpp=cp;
main()
{
    printf("%s",**++cpp);
    printf("%s ",*--*++cpp);
    printf("%s",*cpp[-2]+3);
    printf("%s\n",cpp[-1][-1]+1);
}

```

(11)

```

int a[]={1,3,5,7,9};
int *p[]={a,a+1,a+2,a+3,a+4};
int **pp=p;

```

```

main()
{
    int *x,y;
    printf("%d,%d\n",a,*a);
    printf("%d,%d,%d\n",p,*p,**p);
    printf("%d,%d,%d\n",pp,*pp,**pp);
    x=*++pp;
    printf("%d,%d,%d\n",pp--p,*pp--a,**pp);
    pp=p;
    **pp++;
    printf("%d,%d,%d\n",pp--p,*pp--a,**pp);
    y=*++*pp;
    printf("%d,%d,%d\n",pp--p,*pp--a,**pp);
    ++**pp;
    printf("%d,%d,%d\n",pp--p,*pp--a,**pp);
    printf("%d,%d\n",*x,y);
}

```

(12)

```

main()
{
    float g1(),g2(),sum();
    printf("sum= %.2f\n",sum(g1,g2));
}

float sum(g1,g2)
float (*g1)(),(*g2)();
{
    float a,b,c;
    a=4.8;
    b=5.7;
    c=( *g1)(a,b)+( *g2)(a,b);
    return(c);
}

float g1(x,y)
float x,y;
{
    return(x+y);
}

float g2(x,y)
float x,y;
{
    return(x*y);
}

```

4. 使用指针的处理方法,按下述要求编程,并上机验证。

- (1) 有三个已知的不同的字符串,按由大到小的顺序输出。
- (2) 将一个长度为a的字符串从第n个字符开始的全部字符拷贝到另一个字符数组中。
- (3) 有10个不等长的字符串放在一个指针数组中,对它实现如下操作:

- ① 查找某一个字符串。
- ② 修改某一个字符串。
- ③ 删除某一个字符串。
- ④ 复制某一个字符串。
- ⑤ 排序10个字符串。
- (4) 将一个长度为 n 的字符串,用函数实现其逆序输出。
- (5) 按下列要求输入和输出下列数据阵列:
输入阵列如下:

```

1  2  3  4
5  6  7  8
9 10 11 12

```

输出阵列如下:

```

12 11 10 9
8  7  6  5
4  3  2  1

```

- (6) 求出下列 a 数组的各元素之和。

```
int a[2][3][4];
```

a 数组各元素的值按下述规律赋给:

```
a[i][j][k]=i*3*4+j*4+k;
```

- (7) 自己编写一个字符串比较函数 `strcmp1()`:

```
strcmp1(s1, s2)
```

当 $s1=s2$ 时,返回0;当 $s1>s2$ 时,返回正值;当 $s1<s2$ 时,返回负值。

- (8) 用“起泡”排序法,对一个有 n 个数据项的 `int` 型数据序列进行排序。编程时,可假定 $n=10$ 。

- (9) 有 n 个小孩按顺序号排成一个圆圈。从第1个小孩开始作1至3报数,凡是报数为3的小孩退出圈子,求出最后出圈子的那个小孩是多少号?

- (10) 一个班有8个学生,用学号标识,每人有3门功课的成绩。要求:

- ① 用函数 `input()` 输入8个学生的各门成绩;
- ② 用 `aver()` 函数求出每个学生的平均成绩;
- ③ 用一个函数找出平均分在85以上(含85)的优秀生,并输出它的学号和平均成绩;
- ④ 用一个函数找出平均分在60以下(不含60)的不及格学生,并输出它的学号和平均成绩;
- ⑤ 编写一个程序,使用上述4个函数,对8个学生成绩作如上处理。

第八章 结 构

本章介绍 C 语言中一种重要的构造的数据类型——结构。具有该种类型的变量称为结构变量。结构与数组同是构造的数据类型,但结构不同于数组仅在于它是不同数据类型变量的集合,而数组要求是相同数据类型变量的集合。因此,在某种意义上讲,结构的应用比数组更加广泛。在较复杂的数据结构中,在数据库系统中,结构类型都有着较广泛的应用。本章从结构和结构变量的基本概念讲起,着重介绍它在 C 语言编程中的应用。

8.1 结构的概念

结构与数组都是构造的数据类型,它们都是数目固定的若干个变量的集合,它们在概念上有很多相似之处,学习结构时应与数组比较,这样会更快地掌握结构,记住结构与数组的区别仅在于不同类型变量还是相同类型变量的集合。

8.1.1 结构和结构变量的定义

结构是一种数目固定、类型不同的若干个变量的有序的集合。结构变量是指具有某种结构类型的变量。定义结构变量之前应该先定义某种结构类型,有了这种结构类型后,再定义具有这种结构类型的结构变量、指向结构变量的指针以及结构数组等。结构数组是指数组元素为结构变量的数组。

下面是定义结构类型即结构模式的格式:

```
struct <结构名>{  
    <结构成员说明>  
};
```

其中,struct 是关键字,<结构名>同标识符,<结构成员说明>是给出该结构模式的若干个成员,包含各个成员的名字(即变量名)和类型。C 语言中允许出现的各种类型的变量都可以作结构成员,包含 int 型、float 型、double 型、char 型变量,以及指针、数组、联合变量等。结构变量和指向结构变量的指针也可以作结构的成员,即结构定义是可以嵌套的。但是,某类结构的结构变量可以是另一类结构的结构成员,而不可是本身结构的结构变量。指向结构变量的指针可以是本身结构的成员。

例如:

```
struct card  
{  
    int pips;  
    char suit;  
};
```

这是一个结构模式,其结构名是 card,它有两个成员:一个是 int 型变量 pips,另一个是字

符型变量 `suit`。该结构是用来描述一张扑克牌的, `pips` 表示该牌的点数, 而 `suit` 表示该牌的花色, 其花色用字符 's' (黑桃), 'h' (红桃), 'd' (方块), 'c' (梅花) 来表示。任何一种结构模式都是对某种客观事物的抽象。

在结构模式已定义好后, 可以用下述格式来定义具有某种结构模式的结构变量。

```
struct<结构名><结构变量名表>;
```

其中, <结构名>是已被定义的某种结构模式, <结构变量名表>中可有若干个结构变量、指向结构变量的指针、结构数组等, 它们之间用逗号分隔。例如:

```
struct card c1, c2, *pc, cc[52];
```

其中, `c1`和`c2`是具有 `card` 结构模式的结构变量, `pc` 是指向具有 `card` 结构模式的结构变量的指针, `cc` 是结构数组名, 它有52个元素, 每个元素是一个具有 `card` 结构模式的结构变量。

由此可见, 定义一个结构变量可以分两步, 先定义好一个结构模式, 再定义具有这种结构模式的结构变量。

在书写时, 可以在定义结构模式后, 马上定义结构变量, 其格式如下:

```
struct<结构名>
{
    <结构成员说明>
}<结构变量名表>;
```

例如:

```
struct card
{
    int pips;
    char suit;
} c1, c2, *pc, cc[52];
```

这种连起来定义与前面分开定义是等价的。如果所要定义的结构变量一次性定义完, 即不再出现用这种结构模式来定义新的结构变量, 则可将该结构模式的结构名省略, 即为无名结构。用这种结构只能一次定义所有的结构变量。例如:

```
struct
{
    int pips;
    char suit;
} c1, c2, *pc, cc[52];
```

下面举例说明结构定义的嵌套, 即用另一个结构的结构变量作该结构的成员。

```
struct date
{
    int day, month, year;
    char mon_name[4];
};
struct student
{
    char *name;
    char sex;
```

```

    int old;
    struct date birthday;
};

```

这里有两个结构模式 date 和 student。在 date 结构中,使用了 char 型数组作结构成员,在 student 结构中,使用了 char 型指针作结构成员,又使用结构变量 birthday 作结构成员。

```

struct student Wang, Li, Zhang;

```

其中,Wang, Li, Zhang 是3个具有 student 结构模式的结构变量。该变量将具有 student 结构模式所具有的4个结构成员,其中有一个是结构变量作成员。

下面举一个指向自身结构的指针作结构成员的例子,这又是一种结构定义的嵌套。

```

struct node
{
    int a;
    float b;
    struct node *p;
};

```

其中,p 是指向该结构的结构变量的指针,它作为结构成员,又称指向自身结构的指针作该结构的成员。

最后指出,上面讲过的结构名、结构成员名和结构变量名是三个完全不同的概念,它们允许同名。例如:

```

struct s
{
    int a;
    char b;
    char *s;
} s, m, *pn;

```

这里出现了三个 s,它们分别表示不同的含义。第一次出现的 s 表示结构名,第二次出现的 s 表示某个成员名,该成员是字符指针,第三个 s 表示结构变量名。

8.1.2 结构变量成员的表示

结构变量成员的表示有如下两种方法。

1. 结构变量的成员

结构变量的成员用运算符. 表示。其格式如下:

〈结构变量名〉. 〈结构成员名〉

例如,在前面定义过的结构变量 c1 和 c2,它们是 card 结构模式的结构变量名。

c1.pips 和 c1.suit 分别表示结构变量 c1 的两个成员 pips 和 suit。

同样,c2.pips 和 c2.suit 分别表示结构变量 c1 的两个成员 pips 和 suit。

又例如:

```

struct student
{
    char name[20];

```

```

    int student_no;
    char grade;
} s1, s2;

```

其中,s1和 s2分别是 student 结构模式的两个结构变量。

s1.name 表示 s1的 name 成员;

s1.student_no 表示 s1的 student_no 成员;

s1.grade 表示 s1的 grade 成员。

2. 指向结构变量的指针的成员

指向结构变量的指针的成员用运算符—>表示。其格式如下:

(指向结构变量指针名)—>(结构变量名)

或者

(* <指向结构变量指针名>). <结构变量名>

例如,在前面的定义中,pc 是指向结构模式 card 的结构变量的指针。它的成员表示如下:

pc->pips 表示指针 pc 的成员 pips;

pc->suit 表示指针 pc 的成员 suit。

或者

(* pc). pips 等价于 pc->pips;

(* pc). suit 等价于 pc->suit。

8.1.3 结构变量的赋值

结构变量可以被赋值,也可以被赋初值。一般编译系统要求外部或静态的结构变量才能被赋初值。

给结构变量赋初值的方法与给数组赋初值的方法相似,使用初始值表。例如,给前面已定义过的结构变量 c1和 c2赋初值如下:

```

struct card c1={12,'s'};
struct card c2={1, 'h'};

```

这里,给结构变量 c1赋初值后,c1为黑桃 Q,又给结构变量 c2赋初值后,使 c2为红桃 A。

又例如,前面定义了 student 结构模式,下面给该模式的一个结构变量赋值:

```

struct student s1={"Wang Ping", 7601, 'A'};

```

s1是一个具有 student 结构模式的一个结构变量,它有3个成员,赋初值时使用初始值表对它的三个成员分别赋初值,要求顺序与类型要一致。

给指向结构变量的指针赋初值要用相同类型的地址值。或者将一个相同类型结构变量的地址值赋给指针,或者将一个指向相同类型结构变量的指针赋给它。例如:

```

struct card c1, * pc=&c1;
struct card c1, * pc=&c1, * pd=pc;

```

给结构变量赋值,类似于给数组赋值,即给该结构变量的各个成员赋值,在给结构变量成员赋值时一般要求类型一致。例如:

```

struct student s1, * ps;
s1.name="Li Ping";

```



```

s1.student_no=7602;
s1.grade='B';
ps->name="Zhang han";
ps->student_no=8603;
ps->grade='A';

```

另外,可将一个结构变量整个地赋值给另一个结构变量,但要求两者是同一个结构模式的结构变量,这相当于给所有成员一次赋值。例如:

```

struct card c1, c2={5, 'h'};
c1=c2;

```

这里, `c1` 和 `c2` 是同一个结构模式 `card` 的两个结构变量,并且 `c2` 已被赋了初值。可将 `c2` 赋值给 `c1`,使得 `c1` 的各个成员的值与 `c2` 的相同。

给指向结构变量的指针赋值,可将某个相同类型的结构变量的地址值赋给它,也可以使用 `malloc()` 函数直接给指针分配内存空间。例如:

```

struct student s1, *ps;
ps=&s1;

```

或者

```

ps=(struct student *) malloc (sizeof (struct student));

```

这两种方法都将为指针 `ps` 提供了地址值。第一种方法,使指针 `ps` 指向结构变量 `s1`;第二种方法是使用 `malloc()` 函数直接在内存中分配一个大小为 `student` 结构模式的结构变量的大小的空间。

下面举一个给结构变量赋值的例子。

[例8.1] 结构变量赋值和输出部分成员值。

```

struct point
{
    float x[2];
    struct point *next;
};
main()
{
    struct point p1,p2,*top;
    top=&p1;
    p1.x[0]=1.2;
    top->x[1]=2.3;
    p1.next=&p2;
    p2.x[0]=-3.4;
    p1.next->x[1]=-4.5;
    p2.next=0;
    printf("%.2f, %.2f\n",p1.x[0],p1.x[1]);
    printf("%.2f, %.2f\n",p2.x[0],p2.x[1]);
}

```

执行该程序输出如下结果:

```

1.2, 2.3
-3.4, -4.5

```

说明:

(1) 程序开始定义一个名为 point 的结构模式,它有两个成员:一个是 float 型数组,另一个是指向自身结构的指针。

(2) 在 main()内,定义了3个具有 point 结构模式的结构变量 p1和 p2以及指向结构变量的指针 top。

(3) 分别给指针 top 和两个结构变量赋值。使 top 指向结构变量 p1;又给结构变量 p1的2个成员赋值,数组成员的两个元素值分别为1.2和2.3,指针成员获得结构变量 p2的地址值;结构变量 p2的两个成员中,数组成员的两个元素分别获得-3.4和-4.5,指针成员被赋值为0。三个变量赋值情况如下图8.1所示:

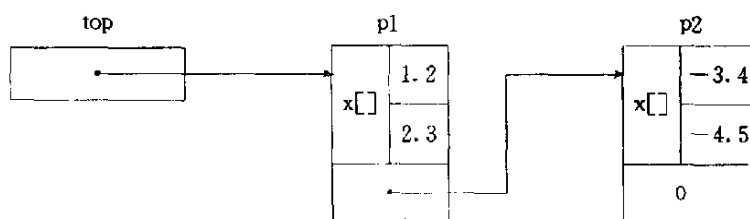


图 8.1

(4) 主函数中又使用两个 printf()函数的语句输出 p1和 p2结构变量中的 x[]数组成员的两个元素值,它们是用结构变量的成员表示的。

8.1.4 结构变量的运算

结构变量的整体上只有赋值运算,即将一个已被赋值(包含赋初值)的结构变量赋给相同类型的另一个结构变量。此外,结构变量的运算主要是指结构变量成员的运算,结构变量的成员具有该成员类型所允许的所有运算。

【例8.2】 结构变量的运算。

```

struct student
{
    char * name;
    long stu_no;
    float math;
    float c_language;
    float english;
};

main()
{
    static struct student s1={"Li jing",9705,89.5,90.0,80.5},
                        s2={"Ma li li",9708,95.0,87.5,84.5};

    float m1,m2;
    m1=(s1.math+s1.c_language+s1.english)/3;
    m2=(s2.math+s2.c_language+s2.english)/3;
    printf("%s:\t%.2f\n",s1.name,m1);
    printf("%s:\t%.2f\n",s2.name,m2);
}

```

执行该程序输出结果如下:

Li jing: 86.67

Ma li li: 89.00

说明：该程序中开始定义了两个结构变量 s1 和 s2，并且赋了初值。在程序中对 s1 和 s2 两个结构变量的成员进行求和等计算。

8.2 结构与数组

数组可以作为结构的成员，在前面的例子中已经看到了。结构变量也可以作数组的元素，这样的数组称为结构数组。

8.2.1 数组作为结构成员

下面是一个数组作为结构成员的例子。

```
struct student
{
    char name[20];
    long student_no;
    float score[3];
} s2, s2, *ps;
```

该结构模式中，有两个数组成员：一个是 char 型的一维数组 name，有 20 个元素；另一个是 float 型的一维数组 score，有 3 个元素。第一个数组成员可改为指针，即用 char * name；代替 char name[20]；。

结构成员是数组时，该数组的各元素相当于该结构的成员。上例中，数组 score[] 各元素的表示如下：以 s1 结构变量为例。

```
s1.score[0]
s1.score[1]
s1.score[2]
```

可以直接给该数组的各元素赋值，如下所示：

```
s1.score[0]=80.0;
s1.score[1]=85.5;
s1.score[2]=91.5;
```

对于具有数组成员的结构变量，可以赋初值，对数组成员的赋值方法与给数组赋初值方法相同。例如：

```
struct student s2={"Wang Li", 9705, {80.0, 85.5, 91.5}};
```

这里，将一个字符串赋初值给 char 型数组 name，将由三个数据项组成的初始值表赋初值给 float 型数组 score。

具有数组成员的结构变量可按结构成员的办法输出数组成员的各元素。例如：

```
printf("%c\n", s2.name[0]);
```

输出字符 'W'。

```
printf("%.2f\n", s2.score[2]);
```

输出浮点数 91.50。

8.2.2 结构数组

结构数组在C语言程序中使用较多。下面通过实例来说明结构数组的应用。

[例8.3] 给52张扑克牌赋值,并且将13张红桃输出显示。

程序是由主函数和三个被调用函数组成,三个被调用函数中 `assign_value()` 用来给扑克牌赋值, `extract_value()` 用来对某张扑克牌析值, `print_value()` 函数用来显示输出某张扑克牌的花色与点数。每张扑克牌用下述结构表示:

```
struct card
{
    int pips;
    char suit;
};
```

52张扑克牌用结构数组表示如下:

```
struct card card[52];
```

程序内容如下:

```
struct card
{
    int pips;
    char suit;
};
main()
{
    struct card card[52];
    int i;
    for(i=0;i<13;i++)
    {
        assign_values(card+i,i+1,'c');
        assign_values(card+i+13,i+1,'d');
        assign_values(card+i+26,i+1,'h');
        assign_values(card+i+39,i+1,'s');
    }
    for(i=0;i<13;i++)
        print_value(card+i+26);
    printf("\n");
}
assign_values(c_ptr,p,s)
struct card * c_ptr;
int p;
char s;
{
    c_ptr->pi ps=p;
    c_ptr->suit=s;
}
extract_value(c_ptr,p_ptr,s_ptr)
struct card * c_ptr;
```

```

int *p_ptr;
char *s_ptr;
{
    *p_ptr=c_ptr->pips;
    *s_ptr=c_ptr->suit;
}
print_value(c_ptr)
struct card *c_ptr;
{
    int p;
    char s, *name;
    extract_value(c_ptr,&p,&s);
    name=(s=='c')?"clubs";(s=='d')?"diamonds";(s=='h')\
        ?"hearts";(s=='s')?"spades":"error";
    printf("\ncard: %d of %s",p,name);
}

```

执行该程序输出如下结果：

```

card: 1 of hearts
card: 2 of hearts
card: 3 of hearts
card: 4 of hearts
card: 5 of hearts
card: 6 of hearts
card: 7 of hearts
card: 8 of hearts
card: 9 of hearts
card: 10 of hearts
card: 11 of hearts
card: 12 of hearts
card: 13 of hearts

```

说明：

该程序中 card[52]是一个结构数组，它有52个元素，每个元素是一个结构变量，它是通过调用 assign_value()函数来赋值的。

在输出13张红桃牌时，调用了 print_value()函数，在该函数中又调用了析值函数 extract_value()用来将一张扑克牌的点数和花色分别存放在两个变量中，并将表示花色的单字符转换为该种花色的英语单词表示，然后输出显示在屏幕上。

在该程序中，由于多次进行函数调用，在调用中使用了传址调用的方式，函数的形参用指向结构变量的指针，关于这一点后面还会详述。

[例8.4] 给定某个月的英文单词表示的前3个字符，输出该月的天数。假定2月为28天。

程序内容如下：

```

struct month
{
    int number_of_day;
    char name[4];
}

```

```

};
main()
{
    int i;
    char *m;
    static struct month months[12]=
        {{31,"Jan"},{28,"Feb"},{31,"Mar"},
         {30,"Apr"},{31,"May"},{30,"Jun"},
         {31,"Jul"},{31,"Aug"},{30,"Sep"},
         {31,"Oct"},{30,"Nov"},{31,"Dec"}}};
    printf("Input month's name (3 characters): ");
    scanf("%s",m);
    for(i=0;i<12;i++)
        if(strcmp(m,months[i].name)!=0)
        {
            printf("%s: %d\n",m,months[i].number_of_day);
            break;
        }
}

```

执行该程序显示如下信息：

Input month's name(3 characters): May ✓

May: 31

[例8.5] 分析下列程序的输出结果。

下列程序中,有结构数组a,还有指针结构数组p,即该数组的元素是指向结构变量的指针。该结构模式又是由指针成员构成的。从该程序中可以看出:指针可作为结构的成员,指向结构变量的指针,又可作为数组元素。此例稍为复杂,为了便于分析,采用图解法。

程序内容:

```

struct s1
{
    char *s;
    struct s1 *s1p;
};
main()
{
    static struct s1 a[]={{"abcd",a+1},
                           {"efgh",a+2},{"ijkl",a}};
    struct s1 *p[3];
    int i;
    for(i=0;i<3;i++)
        p[i]=a[i].s1p;
    printf("%s\t%s\t%s\n",p[0]->s,(*p)->s,(* *p).s);
    swap(*p,a);
    printf("%s\t%s\t%s\n",p[0]->s,(*p)->s,(*p)->s1p->s);
    swap(p[0],p[0]->s1p);
    printf("%s\t",p[0]->s);
    printf("%s\t",(*++p[0]).s);
}

```

```

printf("%s\n", ++(*++(*p)->slp).s);
}
swap(p1,p2)
struct s1 *p1,*p2;
{
    char *temp;
    temp=p1->s;
    p1->s=p2->s;
    p2->s=temp;
}

```

在分析输出结果前,先给出该程序中各个变量赋值后之间的相互关系。图解如图8.2所示:

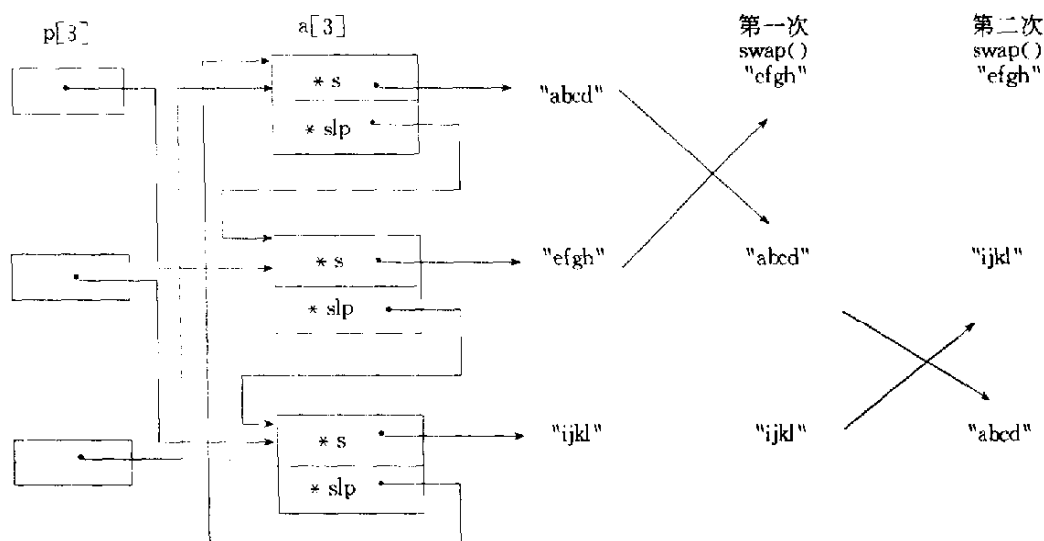


图 8.2

执行该程序输出如下结果:

```

efgh    efgh    efgh
abcd    abcd    ijkl
ijkl    abcd    jkl

```

说明:

(1) 该程序较为复杂,其复杂性表现如下:

- 指针作结构成员。这里有一般的字符指针,还有指向自身结构的指针。
- 结构数组。`a` 是一个一维结构数组名。它有3个元素,每个元素是一个具有 `s1` 结构名的结构变量。该数组被赋初值。
- 指针结构数组。程序中 `p` 是一个指针结构数组,`p` 有3个元素,每个元素是一个指向结构的指针,它是通过 `for` 循环给赋值的,`p[0]` 的地址值为 `a[0].slp`, `p[1]` 的地址值为 `a[1].slp`, `p[2]` 的地址值为 `a[2].slp`。

(2) 根据数组 `a` 和 `p` 被赋值后的关系给出前面的关系图。

(3) 主函数中两次调用 `swap()` 函数。`swap()` 函数的功能是用来交换两个指向结构变量的

指针所指向的结构变量的 s 成员所指向的字符串。第一次调用 swap() 函数时, 实参用 *p, a, 即将 a[1] 的结构变量成员 s 所指向的字符串 "efgh" 与 a[0] 的结构变量成员 s 所指向的字符串 "abcd" 进行交换。第二次调用 swap() 函数时, 实参用 p[0], p[0] -> slp, 即将 p[0] 指向的 a[1] 的结构变量成员 s 所指向的字符串 "abcd" 与 p[0] 的 slp 成员所指向的结构变量 a[2] 的结构变量成员 s 所指向的字符串 "ijkl" 进行交换。

(4) 分析输出结果。

程序中第一个 printf() 函数输出结果是相同的, 即为 efgh。因为 p[0] -> s, 可表示为 (* (p+0)) -> s, 即 (*p) -> s, 而 (*p) -> s 又可表示为 (* *p). s, 所以, 三个参数表达式是等价的, 都是 a[1] 结构变量的 s 所指向的字符串。

第二个 printf() 函数输出中, 前两个参数表达式是等价的, 它们是经过第一次交换后, a[1] 结构变量成员 s 所指向的字符串 abcd。而第三个参数表达式 (*p) -> slp -> s 是 a[2] 结构变量的 s 成员所指向的字符串 ijkl。

第三、四、五个 printf() 函数输出是在第二个交换后进行的。p[0] -> s 是 a[1] 结构变量成员 s 所指向的字符串 ijkl; (* ++p[0]). s 等价于 (++p[0]) -> s, 即 p[1] -> s 是 a[2] 结构变量成员 s 所指向的字符串 abcd; ++(* ++(*p) -> slp). s 等价于 ++(++p[0] -> slp) -> s, 其中, ++p[0] -> slp 是指 a[1], 因为这时 p[0] 已指向 a[2] 了 (前一个 printf() 函数对 p[0] 进行了增1运算), a[2] -> slp 是 a[0], 再增1后为 a[1], 这时 ++a[2] -> s 是一个指向 ijkl 字符串中 j 字符的地址值, 因此, 输出为 jkl。

8.3 结构与函数

结构在函数方面的应用表现在如下两个方面:

- 结构变量和指向结构变量的指针可以作为函数参数。
- 结构变量和指向结构变量的指针可以作为函数的返回值。

8.3.1 结构变量和指向结构变量的指针作为函数参数

1. 结构变量作函数参数

结构变量作函数参数时, 形参和实参都要求是同一种结构模式的结构变量。函数调用时, 系统将实参拷贝一个副本给形参, 这种调用方式在被调用函数中无法改变调用函数的参数。

下面是一个结构变量作函数参数的实例。

[例8.6] 已知两个复数

1.0+i2.0 和 3.0+i4.0

求它们之和与积。

定义表示复数的结构如下:

```
struct complex
{
    float re, im;
};
```

程序内容如下:


```

struct complex
{
    float re,im;
};
#define CMP struct complex
main()
{
    static CMP x={1.0,2.0},y={3.0,4.0};
    CMP z1,z2,add(),multiply();
    z1=add(x,y);
    z2=multiply(x,y);
    printf("(%.2f+i%.2f)+(%.2f+i%.2f)=",
           x.re,x.im,y.re,y.im);
    printf("%.2f+i%.2f\n",z1.re,z1.im);
    printf("(%.2f+i%.2f)*(%.2f+i%.2f)=",
           x.re,x.im,y.re,y.im);
    printf("%.2f+i%.2f\n",z2.re,z2.im);
}
CMP add(x,y)
CMP x,y;
{
    CMP z;
    z.re=x.re+y.re;
    z.im=x.im+y.im;
    return(z);
}
CMP multiply(x,y)
CMP x,y;
{
    CMP z;
    z.re=x.re*y.re-x.im*y.im;
    z.im=x.re*y.im+x.im*y.re;
    return(z);
}

```

执行该程序输出结果如下:

$(1.00+i2.00)+(3.00+i4.00)=4.00+i6.00$

$(1.00+i2.00)*(3.00+i4.00)=-5.00+i10.00$

说明: 该程序由一个 main() 函数和两个被调用函数 add() 和 multiply() 组成。两个被调用函数的形参都是结构变量, 结构变量可作为函数的参数。另外, 这两个函数的返回值又都是结构变量, 结构变量可作为函数的返回值, 这种函数称为结构函数。

2. 指向结构变量的指针作函数参数

结构变量可以作为函数的参数, 前面举过的例子中已经出现。指向结构变量的指针作函数的形参, 要求对应的调用函数的实参用相同类型的结构变量的地址值, 实现传址调用, 在被调用函数中可通过改变形参指针所指向结构变量的值来改变实参的值。另外, 这种传址调用比用结构变量作函数参数实现的传值调用具有更高的效率。因为传值调用时, 系统将拷贝实参的副

本给被调用函数的形参,而传址调用时,只传递其地址值,特别是当结构变量比较复杂时,传值调用要花费较多的时间和占用较大的空间,因此效率较低。因此,在实际应用中较多地使用指向结构变量的指针作函数参数。

[例8.7] 已知今天的日期(包含年、月、日)输出明天的日期。

程序内容如下:

```
struct ydate
{
    int day, month, year;
};
main()
{
    struct ydate today, tomorrow;
    printf("Enter today's date(yyyy/mm/dd);\n");
    scanf("%d/%d/%d", &today.year, &today.month, &today.day);
    if(today.day != number_of_days(&today))
    {
        tomorrow.day = ++today.day;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if(today.month == 12)
    {
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else
    {
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }
    printf("Tomorrow's date is %d/%d/%d\n",
        tomorrow.year, tomorrow.month, tomorrow.day);
}

is_leap_year(pd)
struct ydate *pd;
{
    int leap_year = 0;
    if ((pd->year % 4 == 0 && pd->year % 100 != 0) || pd->year % 400 == 0)
        leap_year = 1;
    return(leap_year);
}

number_of_days(pd)
struct ydate *pd;
{
    int day;
```

```

static int days_ptr_month[13]=
    {0,31,28,31,30,31,30,31,31,30,31,30,31};
if (is_leap_year(pd)&&(pd->month==2))
    day=29;
else
    day=days_ptr_month[pd->month];
return(day);
}

```

执行该程序,提示如下信息:

Enter today's date(yyyy/mm/dd):

1997/12/31 ✓

输出如下信息:

Tomorrow's date is 1998/1/1

说明: 该程序中,在被调用函数 `is_leap_year()` 和 `number_of_days()` 中都使用了指向结构变量的指针作函数形参,调用时所对应的实参为相同结构类型的结构变量的地址值。在这两个函数中,使用指向结构变量的指针都不是为了改变调用函数中的实参值,而是为提高函数调用时传递数据的效率。

8.3.2 结构变量和指向结构变量的指针作函数返回值

结构变量和指向结构变量的指针都可以用作函数的返回值。结构变量作函数返回值,该函数称为结构函数。

1. 结构函数

在前面讲过的例8.6中,出现了两个结构函数 `add()` 和 `multiply()`。这两个函数的返回值都是结构名为 `complex` 的结构变量。

结构函数在应用中也是常出现的,由于前面已举过例子,这里不再详述。

2. 指向结构变量的指针作函数返回值

函数的返回值可以是结构变量,也可以是指向结构变量的指针。

下面举一个函数返回值是指向结构变量的指针的例子。

[例8.8] 查找某个学生的成绩,描述学生的结构模式如下:

```

struct student
{
    char * name;
    float score [3];
};

```

其中, `name` 用来存放学生的名字, `score` 是存放学生三门功课成绩的 `float` 型数组。

程序内容如下:

```

struct student
{
    char * name;
    float score [3];
} stu[5] = {"Li", {90, 87, 83}, "Ma", {98, 85, 78}, "Wang", {96, 89, 87},

```

```

        "Huang", {80, 82, 72}, "Zhang", {88, 75, 68} };

main()
{
    struct student *s1, *find();
    s1=find(stu);
    printf(" %s: %.2f, %.2f, %.2f\n", s1->name, s1->score[0],
        s1->score[1], s1->score[2]);
}

struct student *find(s)
struct student s[];
{
    int i;
    char name1[20];
    printf("Input student's name: ");
    scanf(" %s", name1);
    for(i=0; i<5; i++)
        if (strcmp(name1, s[i].name) == 0)
            return(s+i);
}

```

执行该程序,显示如下信息:

```

Input student's name: Zhang
Zhang: 88.00, 75.00, 68.00

```

说明: 该程序中,函数 find()的返回值是一个指向结构变量的指针。它所返回的地址值在主函数中赋给结构变量指针 s1。通过函数的返回值达到函数之间的信息传递。

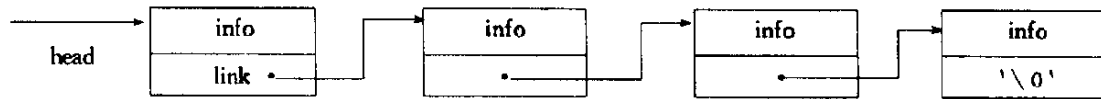
8.4 链 表

8.4.1 链表的概念

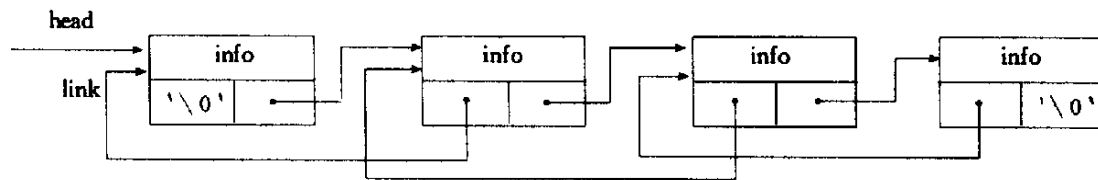
链表是一种常用的数据结构。它是一种动态地进行存储分配的数据结构。它不同于数组,它不必事先确定好元素的个数,它可以根据当时的需要来开辟内存单元,它的各个元素不要求顺序存放,因此,它克服使用数组存放数据的一些不足。

链表是由若干个称为结点的元素构成的。每个结点包含有数据字段和链接字段。数据字段是用来存放结点的数据项的;链接字段是用来存放该结点指向另一结点的指针的。每个链表都有一个“头指针”,它是存放该链表的起始地址,即指向该链表的起始结点,它是识别链表的标志,对某个链表进行操作,首先要知道该链表的头指针。链表的最后一个结点,称为“表尾”,它不再指向任何后继结点,表示链表的结束,该结点中链接字段指向后继结点的指针存放 NULL。

链表可分为单向链表和双向链表。两者的区别仅在于结点的链接字段中,单向链表仅有一个指向后继结点的指针,而双向链表有两个指针,一个指向后继结点,另一个指向前驱结点。下图给出单向链表与双向链表的区别。



单向链表



双向链表

链表这种数据结构必须用指针来实现,每个结点中链接字段要包含一个或两个指针,存放与其链接的结点地址。下面是单向链表和双向链表中结点的结构例子。

```
struct link1
{
    char c1[100];
    struct link1 * next;
};
struct link2
{
    char c2[100];
    struct link2 * next;
    struct link2 * prior;
};
```

其中,link1是单向链表中结点的结构名,link2是双向链表中结点的结构名。链表中的结点就具有这种结构名的结构变量。这里仅是一个例子,实际上结点结构的数据字段比这还要复杂些。

由于单向链表比较简单些,下面只讨论单向链表的操作。

8.4.2 链表的操作

这里只讨论单向链表的操作。常用的链表操作有如下几种:

1. 链表的建立

链表建立是在确定了链表结点的结构之后给链表中的若干个结点输入数据。

2. 链表的输出

链表的输出是将一个已建立好的链表中各个结点的数据字段部分地或全部地输出显示。

3. 链表的删除

链表的删除是指从已知链表中按指定关键字删除一个或若干个结点。

4. 链表的插入

链表的插入是指将一个已知结点插入到已知链表中。插入时要指出按结点中哪一个数据

字段进行插入,插入前一般要对已知链表按插入的数据字段进行排序。

5. 链表的存储

该操作是将一个已知的链表存储到磁盘文件中进行保存。

6. 链表的装入

该操作是将已存放在磁盘中的链表文件装入到内存中。

下面将通过6个例子来讲述上述6种操作如何实现。

这里使用一个学生成绩表,该表是由若干个学生的成绩组成,每个结点是一个学生的成绩。为了简化结点,突出链表的操作。定义结点结构如下:

```
struct student
{
    char * name;
    long num;
    int score;
    struct student * next;
};
```

该结构中,有三个成员作为数据字段,其中学号作为关键字,自身引用的指针作为链接字段,用来指出下一个结点位置。为简化结构内容只选用一门课程的成绩。

[例8.9] 建立链表函数。

建立链表函数的算法如下:

设置三个结构指针 head, p 和 q。先用存储分配函数 malloc() 开辟一个结点,并且使指针 p 和 q 分别指向所开辟的结点。再从键盘上读入一个学生的数据赋给 p 所指向的结点。假定学号 num 值为 0 时,链表建立结束,该结点不链入表中。输入第一个结点的数据后,将 p 赋给 head,使 head 指向链表中的第一个结点。如图 8.3(a) 所示。接着,再开辟第二个结点,使 p 指向新结点,读入新结点的数据,并链入新结点,将 p 的值赋给 q->next,使第一个结点的 next 成员指向第二个结点。接着,使 q=p,即 q 将指向新结点。如图 8.3(b) 所示。

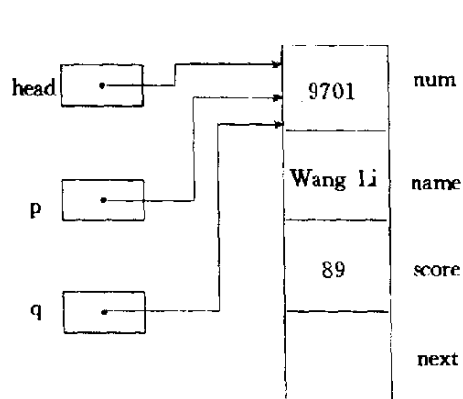


图 8.3 (a)

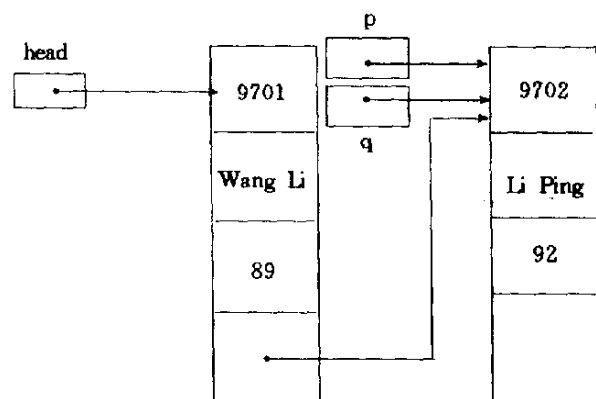


图 8.3 (b)

再用 p 去开辟新结点,赋值后,在学号 num 成员不为 0 的情况下,链入新结点, ..., 直到链

入最后一个新结点为止。这里,设置的三个结构指针,除了 head 作为该链表的头指针之外,p 总是用来指向开辟的新结点的指针,而 q 总是指向 p 所指向的新结点的前一个结点的指针,使用将 p 赋值给 q->next 来链入新结点。

建立链表函数内容如下:

```
struct student *creat_list()
{
    struct student *head, *p, *q;
    head=NULL;
    p=q=(struct student *)malloc(sizeof(struct student));
    n=0;
    printf("Input node data: ");
    scanf("%ld%s%d",&p->num,p->name,&p->score);
    while(p->num!=0)
    {
        n++;
        if(n==1)
        {
            head=p;
            head->next=NULL;
        }
        else
        {
            q->next=p;
            q=p;
            p=(struct student *)malloc(sizeof(struct student));
            printf("Input node data: ");
            scanf("%ld%s%d",&p->num,p->name,&p->score);
        }
        q->next=NULL;
    }
    return(head);
}
```

该函数前定义 n 为一个外部变量,用来记录结点的个数。该函数是一个指针函数,因为它返回一个指向结构变量的指针。所返回的指针为该链表的头指针。头指针是一个链表的标志,链表的许多操作都是从头指针开始的。

[例8.10] 链表输出函数。

链表输出函数是将链表中从头到尾各结点的数据输出显示。

链表输出函数算法如下:

首先要知道输出链表的头指针。假定一个指向链表结点个数的结构指针 p,使它指向链表的第一个结点,即将头指针赋给它,输出 p 所指结点的数据内容。再使 p 后移一个结点,输出该结点数据,直到链表的最后一个结点为止。

链表输出函数内容如下:

```
void print_list(head)
struct student *head;
{
    struct student *p;
```

```

printf("\nNow, There %d node data are :\n", n);
p = head;
if (head != NULL)
do {
    printf(" %ld, %s, %d\n", p->num, p->name, p->score);
    p = p->next;
}while(p != NULL);
else
    printf("There are a empty list!\n");
}

```

说明：这里 n 是一个用来存放链表结点个数的外部变量。

实际中在输出某个结点的数据时，可以根据需要输出结点的部分数据。

该函数形参 $head$ 是用来接收实参传递来的该链表的头指针，可见链表输出函数是用来输出某个链表的从头开始的各个结点的数据。

[例8.11] 链表删除函数。

该函数是用来删除已知链表中的某个结点的。删除某个结点时要给定该结点的关键字，并在结点中找到与给定关键字相匹配的结点，然后将其从链表中删除。

链表删除函数的算法如下：

假定以结点数据中的学号为关键字，设置两个指针 p 和 q 。先使 p 指向链表中的开始结点，并检查该结点是否是要删除的结点。如果不是要删除的结点，则将 p 指向下一个结点，并在此之前将 p 值赋给 q ，使 q 指向刚刚检查过的结点。如果 p 所指向的下一个结点还不是要删除的结点，则使 q 指向该结点， p 再往后移，直到 p 所指向的结点是要删除的结点或整个链表中找不到要删除的结点为止。如果找到了要删除的结点，则分如下两种情况进行处理：

(1) 要删除的结点为头结点时，即 $p == head$ ，则需将 $p->next$ 赋给 $head$ ，即让头指针指向链表中的第二个结点，这时第一个结点被“丢失”。

(2) 要删除的结点不是头结点时，则将 $p->next$ 赋值给 $q->next$ 。因为这时 p 是指向要删除的结点，而 q 指向 p 的前面一个结点，这样做就使 p 前面的一个结点跳过了 p 结点指向了 p 的下一个结点，于是将 p 所指向的要删除的结点“丢失”了。

链表删除函数内容如下：

```

struct student *delete_list(head)
struct student *head;
{
    struct student *p, *q;
    if (head == NULL)
        printf("list null!\n");
    else
    {
        p = head;
        while (num != p->num && p->next != NULL)
        {
            q = p;
            p = p->next;
        }
    }
}

```



```

        if(num == p->num)
        {
            if(p == head)
                head = p->next;
            else
                q->next = p->next;
            printf("delete node: %ld\n", num);
            n--;
        }
        else
            printf("%ld not been found!\n", num);
    }
    return(head);
}

```

说明：该函数是一个指针函数，它返回链表的头指针。

该函数中的 n 是外部变量用来存放该链表的结点个数的。

[例8.12] 链表插入函数。

该函数是用来将一个已知的链表结点插入到已有的链表中。插入结点的操作中，应知道该插入的结点中某个关键字，以便按其关键字的大小顺序插入到适当的位置。

链表插入函数的算法如下：

假定插入结点的关键字为学号 num ，即按学号顺序插入该结点，因此要求已有的链表中结点应按学号的大小排序，假定排序是按升序进行的。插入结点的操作包含两大步骤：一是找到插入结点的位置，二是将结点插入到找到的位置上。设置三个结构指针 s, p, q 。用 s 指向待插入的结点， p 指向链表首结点。将 $s \rightarrow num$ 与 $p \rightarrow num$ 比较，如果 $s \rightarrow num > p \rightarrow num$ ，则将 p 后移，并使用 q 指向刚才 p 指向的结点，继续比较和使 p 后移，直到 $s \rightarrow num \leq p \rightarrow num$ 为止，即找到了 s 插入的位置，将 s 插入到 p 所指向的结点之前。如果 p 所指向的结点已是表尾结点，则 p 不再后移。三种不同插入位置的方法如下：

(1) 插入位置在第一个结点之前，则将 s 赋给 $head$ ，并将 p 赋给 $s \rightarrow next$ 。

(2) 插入位置在表尾之后，则将 s 赋给 $p \rightarrow next$ ，并将 $NULL$ 赋给 $s \rightarrow next$ 。

(3) 插入位置不为(1)和(2)情况时，则将 s 赋给 $q \rightarrow next$ ，即使 q 指向待插入的结点，再将 p 值赋给 $s \rightarrow next$ ，于是在 q 与 p 所指向的两个结点之间插入新结点。

链表插入函数内容如下：

```

struct student *insert_list(head, newstu)
struct student *head, *newstu;
{
    struct student *s, *p, *q;
    p = head;
    s = newstu;
    if(head == NULL)
    {
        head = s;
        s->next = NULL;
    }
}

```

```

else
    while((s->num>p->num)&&(p->next!=NULL))
    {
        q=p;
        p=p->next;
    }
    if(s->num<=p->num)
    {
        if(head==p)
            head=s;
        else
        {
            q->next=s;
            s->next=p;
        }
    }
    else
    {
        p->next=s;
        s->next=NULL;
    }
    n++;
    return(head);
}

```

说明：该函数也是一个指针函数，返回链表的头指针。函数开始时，先判断一下是否是空表。如果是空表，则将插入结点作为该表首结点；如果不是空表，再寻找结点应该插入的位置，其方法是通过后移 p 指针，直到找到位置或移至表尾结点为止。接着，使用 if-else 语句将插入结点的三种情况都包含进去了。即插入在链表的头、中间和尾三种情况。

[例8.13] 链表存储函数。

该函数的功能是将一个已建好的链表中的各个结点数据存储到磁盘文件中，以备后用。

链表存储函数的算法如下：

设置两个指针：head 和 p。用 head 指向链表的头结点，用移动 p 指针来指向链表中的各个结点。打开一个写入结点的数据文件，从头开始每次写入文件中一个结点的信息，直到所有结点写完为止。

链表存储函数内容如下：

```

void save_list()
{
    struct student *p;
    FILE *fp;
    if((fp=fopen("nodefile.txt","w"))==NULL)
    {
        printf("\tnodefile can't open!\n");
        exit(1);
    }
    p=head;

```

```

while(p)
{
    fwrite(p,sizeof(struct student),1,fp);
    p=p->next;
}
fclose(fp);
}

```

说明：该函数无返回值，有一个形参 head，用来接收实参传递过来的头指针。使用该函数时，应包含文件 stdio.h，文件指针 FILE 定义在 stdio.h 中。函数开始先打开一个用来存放链表结点数据的磁盘文件 nodefile，然后通过 while 循环，从头到尾将链表中各结点数据用 fwrite() 函数写到指定的文件中，每次循环写入一个结点数据，直到写完为止。

[例8.14] 链表装入函数。

该函数的功能是将磁盘中的链表文件读入内存，供操作使用。

链表装入函数的算法如下：

打开磁盘中的待装入的磁盘文件，每次从文件中读出一个结点内容存放在指定的内存地址中，直到待装入的磁盘文件中链表文件的所有结点都读入到内存为止。设置三个结构指针：head、p 和 q。用 head 存放链表头指针，p 指向新开辟的存放结点的内存单元，q 指向刚读完信息的结点。

链表函数的内容如下：

```

struct student * load_list()
{
    struct student * p, * q;
    FILE * fp;
    if((fp=fopen("nodefile.txt","r"))==NULL)
    {
        printf("nodefile can't open!");
        exit(2);
    }
    head=(struct student *)malloc(sizeof(struct student));
    if(!head)
    {
        printf("memory is full!");
        exit(2);
    }
    p=q=head;
    while(!feof(fp))
    {
        fread(p,sizeof(struct student),1,fp);
        q=p;
        if(p->next==NULL)
            break;
        p=(struct student *)malloc(sizeof(struct student));
        if(!p)
        {
            printf("memory is full!\n");

```

```

        exit(3);
    }
    q->next=p;
}
q->next=NULL;
fclose(fp);
return(head);
}

```

说明：该函数也是一个指针函数，其返回值是链表的头指针。该函数无形参。

函数中先打开链表文件，通过使用 malloc() 函数不断地开辟内存单元来存放结点数据，将链表文件中各个结点数据都读入内存。函数最后返回头指针。

以上列举了6个有关链表操作的函数。此外，有关链表操作还有其他函数，例如，检索函数，或者通过学号，或者通过姓名来查找一个学生的成绩；修改函数，通过该函数可以修改某个学生的成绩，或者修改一个结点的其他数据，等等。读者有兴趣可以编写出这些函数。

下面将前面讲过的这些函数编写一个完整的程序，实现对链表的某些操作。为了节省篇幅，前面已写出的函数内容就不再重复书写了，读者要想运行该程序应将前面定义的函数内容写到该程序中，该程序见下面的例8.15。

[例8.15] 使用单向链表编写学生成绩管理程序。该程序中只编写了部分管理操作，并且被调用函数前面已有的这里不再重写。

程序内容如下：

```

#include <stdio.h>
#include <conio.h>
int n;
long num;
struct student
{
    long num;
    char *name;
    int score;
    struct student *next;
} *head=NULL, *stu;
void print_list(), save_list();
struct student *creat_list(), *load_list(), *insert_list(), *delete_list();

main()
{
    while(1)
    {
        char s[10];
        clrscr();
        gotoxy(0,0);
        switch(menu_select())
        {
            case 1: head=creat_list();

```

```

        break;
    case 2: print_list(head);
        printf("\n\tPress any key continue!");
        getch();
        break;
    case 3: printf("Input num: ");
        scanf("%ld",&num);
        head=delete_list(head);
        printf("\n\tPress any key continue!");
        getch();
        break;
    case 4: stu=(struct student *)malloc(sizeof(struct student));
        printf("Input insert node data: ");
        scanf("%ld%s%ld",&stu->num,stu->name,&stu->score);
        head=insert_list(head,stu);
        break;
    case 5: save_list();
        printf("\nSave complete,press any key continue!\n");
        getch();
        break;
    case 6: free(head);
        head=load_list();
        print_list(head);
        printf("\n\tPress any key continue!\n");
        getch();
        break;
    case 7: exit(0);
}
}
}

menu_select()
{
    char d[5];
    int c;
    printf("\n\n%16s","");
    printf(" * * * * MENU * * * * \n");
    printf("\t\t1. Creat list\n");
    printf("\t\t2. Display list\n");
    printf("\t\t3. Delete list\n");
    printf("\t\t4. Insert list\n");
    printf("\t\t5. Save list\n");
    printf("\t\t6. Load list\n");

```

```

printf("\t\t7. Quit\n");
do {
    printf("\n\tInput select(1-7): ");
    gets(d);
    c=atoi(d);
} while(c<0||c>7);
return(c);
}

```

该程序还包含如下函数,为节省篇幅不再重写。

```

void print_list()
void save_list()
struct student * creat_list()
struct student * load_list()
struct student * delete_list()
struct student * insert_list()

```

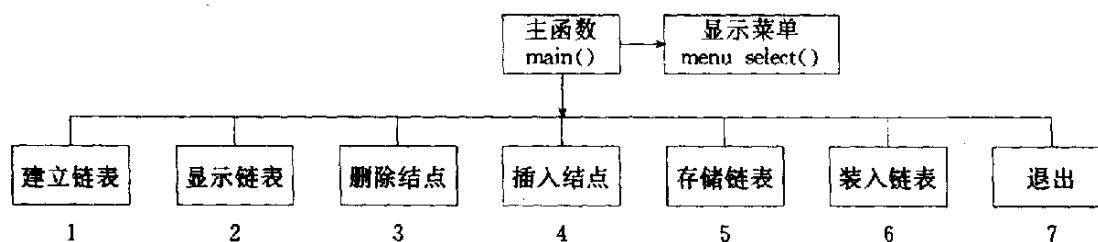
该程序中,只写了主函数 main() 和 menu_select() 函数。在 main() 中通过调用各个被调用函数来实现该程序的功能。在 menu_select() 函数中,给出显示的屏幕菜单和选择菜单的功能。执行该程序显示菜单如下:

*****MENU*****

1. Creat list
2. Display list
3. Delete list
4. Insert list
5. Save list
6. Load list
7. Quit

Input select (1-7):

通过选择不同数字(1-7),可以实现该程序的不同功能。该程序的各功能模块框图如下:



说明:

- (1) 执行该程序,屏幕上显示出上述菜单,供选择使用。
- (2) 选择1后,屏幕显示如下提示信息:

Input noda data:

依次输入学号、姓名和成绩,三个数据项间用空格分隔,回车后,又显示上述提示信息,要求输入下一个结点的三项数据。如果要结束结点数据的输入,请键入0 0 0 ↵。

(3) 选择2后,屏幕上显示当前链表的所有结点数据,每行显示一个结点的三个数据。

(4) 选择3后,屏幕上显示如下信息:

Input num:

要求输入要删除结点的学号数据,输入某个学号回车后,显示出如下信息:

delete node: <学号>

或者

List null!

或者

<学号>not been found!

最后显示:

Press any key continue!

(5) 选择4后,屏幕上显示如下信息:

Input insert node date:

输入要插入结构的数据字段后,按回车键。则该结点被插入。

可以使用该功能建立一个顺序链表。由于该链表是按插入方法生成的,因此,它是排好序的一种链表。

(6) 选择5后,将当前链表存储到一个指定的文件中,备以后调用。存储完成后,显示如下信息:

Save complete, press any key continue!

(7) 选择6后,将指定文件(nodefile.txt)装入内存,并且将其内容显示在屏幕上。该链表可供修改及查询使用。

最后显示如下信息:

Press any key continue!

(8) 选择7后,则退出该程序。

8.5 位 段

位段是一种对信息进行的压缩的方法,信息经压缩后可以节省内存空间。

8.5.1 位段的概念

位段是一种压缩信息的方法,它所使用的是一种结构的数据形式。该方法是在结构中定义一种特殊的成员,该成员是以位为单位定义长度的。采用这种压缩信息方法的好处在于对各数值的存取可采用结构成员的存取方法,操作起来十分方便。

位段的定义格式如下:

在一个结构中可以有若干个被定义的位段式的成员,其格式如下:

unsigned <成员名>:<二进制位数>

位段式成员一般为 unsigned int 型的,<成员名>同标识符,即与一般结构成员名命名法相同。冒号后面写有该位段的长度,以二进制位为单位。例如,

```
struct data_struct
```

```

{
    unsigned f1: 1;
    unsigned f2: 1;
    unsigned f3: 1;
    unsigned opcode1: 3;
    unsigned opcode2: 4;
    unsigned opcode3: 6;
    int a;
};

```

这是一个具有7个成员的结构模式,结构名为 data_struct。前三个成员都是只占1位内存空间,接着三个成员分别占3,4和6位内存空间,最后一个成员不是位段形式,它是一个 int 型变量在16位机中占2个字节。具有这种结构的结构变量共占内存4个字节。

下面定义一个结构变量 d1,其格式如下:

```
struct data_struct d1;
```

按结构成员的方法对其每个位段式成员进行操作。例如,将6赋给 opcode1,将8赋给 opcode2可用如下方式:

```

d1.opcode1=6;
d1.opcode2=8;

```

再将 f1和 f2成员的值(假定 f1和 f2中已被赋值)取出存放在变量 m 和 n 中,可用下列操作:

```

m=d1.f1;
n=d1.f2;

```

可见,对位段式变量进行操作十分方便。

这里,需要注意的是位段所允许的最大数值范围。例如,下面操作

```
d1.opcode1=8;
```

是错误的。因为成员 opcode1的长度是3个二进制位,它的最大取值为二进制数111,即十进制数7。如果将8赋给 d1.opcode1,则 opcode1只能取后面3位,即000了。

又例如,将扑克牌的花色和点数用位段形式可表示如下:

```

struct pcard
{
    unsigned pips: 4;
    unsigned suit: 2;
};

```

其中,pcard 是结构名,它有二个成员,其一是 pips,它是占有4位的位段,用来表示扑克牌的点数,从1至13;其二是 suit,它是占2位的位段,用来表示扑克牌的花色,即梅花、方块、红心和黑桃分别用0,1,2,3表示。t 是一个结构变量。可用如下操作给 t 变量赋值:

```

t.pips=9;
t.suit=2;

```

于是,t 便是扑克牌红心9。另外,位段可以作为结构成员进行如下操作。


```

p=t.pips+2;
if ( t.suit==2)
    printf("hearts");

```

等等。

8.5.2 使用位段时应注意的事项

(1) 位段在内存中的分配方向与机器有关。有的机器从左向右分配,即从高字节向低字节分配,有的机器从右向左分配(如 PDP 机),即从低字节向高字节分配。使用时应注意首尾相连接的问题。

(2) 位段一般采用 unsigned int 型,在位段的同一结构中还可包含不同类型的其他数据类型项。例如,

```

struct table
{
    int count;
    unsigned f1: 1;
    unsigned f2: 1;
    unsigned f3: 1;
    char c;
};

```

该结构中除了含有3个位段成员外,还有一个 int 型成员 count 和一个 char 型成员 c。

(3) 在位段结构中可以定义无名位段,它可以用来作为位段的分隔。例如:

```

struct entry
{
    unsigned type: 4;
    unsigned : 4;
    unsigned count: 8;
};

```

该结构中有三个位段成员,其中有一个无名位段,它占4位,它的作用是将 type 成员和 count 成员用4个空位分隔开,即在 type 成员后面空4位不用,接着再是 count 成员占8位。

(4) 长度为0的位段用来使字边界对齐。使用长度为0的位段可使下一个位段从一个新的字开始存放。例如,

```

struct x
{
    unsigned a: 2;
    unsigned b: 8;
    unsigned : 0;
    unsigned c: 4;
    unsigned d: 8;
}

```

该结构中有5个位段成员,其中有一个长度为0的无名位段,它使得 a 和 b 成员存放在一个字中,但是没有占满,而 c 和 d 成员存放在另一个字中。

(5) 一个位段不能跨越两个字,只能存放在同一个字中。如果前面的几个位段几乎占满一个字,空下来的不足以再放一个位段时,该位段只好从下一个字开始。例如,在16位机中有如下定义:

```
struct y
{
    unsigned f1: 8;
    unsigned f2: 4;
    unsigned f3: 6;
    int a;
    unsigned f4: 8;
}
```

该结构有5个成员,其中有4个位段成员和一个 int 型成员 a。这里, f1 和 f2 成员共12位占一个字, f3 成员占另一个字,因为 f1 和 f2 再加上 f3 共18位,超出一个字。a 占一个字,最后 f4 占一个字,共占4个字。

(6) 不能构造位段数组,也不能对位段变量进行地址操作。

(7) 位段可以用整型格式输出,即可用 %d, %o 和 %x 等格式符进行输出。位段可在表达式中引用,系统自动将位段转换成 int 型数。

(8) 位段不能定义在联合中,位段也不能作为函数的返回值。

练 习 题

1. 什么是结构?它与数组有何相同之点和不同之处?
2. 如何定义一个结构变量?定义结构变量时必须指出它是哪种结构模式的结构变量吗?
3. 什么是指向结构变量的指针?如何给它赋值?
4. 什么是结构数组?如何给它赋初值?
5. 结构变量的成员如何表示?指向结构变量的指针的成员如何表示?
6. “结构的定义可以嵌套”这句话的含意是什么?
7. 结构变量如何赋初值?如何赋值?
8. 结构变量有哪些运算?
9. 举例说明如何将较复杂的结构关系用图解法表示?这种方法有何好处?
10. 结构变量和指向结构变量的指针都可以作为函数参数,二者有何不同?
11. 什么是结构函数?举例说明有哪些结构函数?
12. 什么是链表?什么是单向链表?如何用结构表示链表?链表有哪些操作?
13. 什么是位段?为什么要使用位段?在使用位段中应注意哪些事项?

作 业 题

1. 判断下列描述是否正确,对者划√,错者划×。
 - (1) 结构和数组的区别仅在于结构的成员可以是不同类型,而数组的元素只能是相同类型。
 - (2) 结构变量和指向结构变量的指针的成员的表示是不同的。
 - (3) 在同一结构中,结构名、结构成员名和结构变量名是不允许相同的。


```

        char suit;
    } c1;
main()
{   c1={2, 's'};
    printf("%d, %c\n", c1);
}

```

(2)

```

        :
struct complex
{
    double re, im;
} d1, d2;
scanf("%lf%lf", &d1);
d2=d1;
    :

```

(3)

```

struct s
{
    int a;
    char * name;
} s1, * ps;
ps->name="blue";
ps->a=5;
s1=ps;
    :

```

(4)

```

struct m
{
    int a[5];
    float b;
} * m1={{1,2,3,},3.16};
struct m m2={3.5,{5,6,7,8,9}};
    :

```

(5)

```

struct abc
{
    int xy;
    struct abc mn;
} s1, s2, * ps;
ps=s1;
ps.xy=56;

```

4. 分析下列程序的输出结果。

(1)

```

struct student
{
    char * name;

```

```

    float score;
} * p;
main()
{
    struct student a;
    p=&a;
    p->score=90.5;
    p->name=(char *)malloc(50);
    strcpy(p->name,"Li Ping");
    printf("%s\t%.2f\n",p->name,(*p).score);
}

```

(2)

```

struct s1
{
    char *s;
    int i;
    struct s1 *slp;
};
main()
{
    static struct s1 a[]={{ "abcd",1,a+1},
                          {"efgh",2,a+2}, {"ijkl",3,a}};

    struct s1 *p=a;
    int i;
    printf("%s\t%s\t%s\n",a[0].s,p->s,a[2].slp->s);
    for(i=0;i<2;i++)
    {
        printf("%d\t",--a[i].i);
        printf("%c\n",++a[i].s[3]);
    }
    printf("%s\t",++(p->s));
    printf("%s\t",a[(++p)->i].s);
    printf("%s\n",a[--(p->slp->i)].s);
}

```

(3)

```

struct student
{
    int num;
    char name[30];
    float score;
};
main()
{
    static struct student s[]={ {7001,"Li",87.5},
                                {7002,"Wang",90.0}, {7003,"Han",90.5},
                                {7004,"Huang",75.5}, {7005,"Lu",81.5}};

    fun(s+1);
}
fun(s)

```

```

struct student *s;
{
    printf("%s\t%.2f\n",s->name,(*s).score);
}

```

```

(4)
struct s
{
    int a,*b;
} *p;
int x[]={11,12},y[]={21,22};
main()
{
    static struct s a[]={10,x,20,y};
    p=a;
    printf("%d",*p->b);
    printf("%d",(*p).a);
    printf("%d",++p->a);
    printf("%d",++(*p).a);
    printf("%d\n",(*p).a+*(*p).b);
}

```

```

(5)
struct {
    unsigned b1:4;
    unsigned b2:3;
    unsigned b3:5;
    unsigned :0;
    unsigned b4:8;
    unsigned b5:4;
    unsigned :4;
    unsigned b6:12;
}s;

main()
{
    unsigned *p=(unsigned *)&s;
    s.b1=10;
    s.b2=6;
    s.b3=5;
    s.b4=0xaa;
    s.b5=0x15b;
    printf("%x\n",*p);
    printf("%x\n",*(p+1));
    printf("%x\n",*(p+2));
    printf("%x,%x,%x\n",s.b1,s.b3,s.b5);
}

```

5. 使用结构变量编写下述程序。

(1) 通过定义一个结构变量(含年、月、日成员),指定某个日期后计算该日在本年中是第几天?考虑闰年问题。

(2) 使用下列万年历公式,计算出今年你的生日是星期几?

$$s=y-1+\frac{y-1}{4}+\frac{y-1}{400}-\frac{y-1}{100}+d$$

其中,y 是公元号,d 是从元旦起到指定的那天之间的总天数。

(3) 有8个学生,每个学生的数据包含有学号,姓名,年龄,4门功课的成绩。要求输出:

① 每个学生的姓名和4门功课的平均成绩。

② 输出分数最高学生的姓名和平均成绩。

(4) 建立一个链表,链表中每个结点包含姓名、年龄、电话号码。编程按姓名查找某个人的电话号码。

(5) 编程将两个结点内容完全相同的链表合并为一个链表。

(6) 将一个已知的链表按逆序排列,即将原表头作新表的表尾,原表的表尾作新表的表头。

(7) 有100个小孩围成一圈,给他们从1开始依次编号,假定从第8个小孩起进行1至3报数,当报到“3”时,该小孩出列,剩下的仍围成一圈,接着继续进行1至3报数,报到“3”的小孩出列,以次继续下去,求出最后一个小孩的编号。

(8) 某书店出售下列各种计算机书籍,其书名、数量、单价如下所示:

BASIC	50	8.50
PASCAL	40	11.30
COBOL	45	9.70
dBASE	55	8.90
C	70	15.50

按下列要求编程:

① 显示各种书籍的书名、数量和单价。

② 按书名查找某种书的数量。

③ 给定书名和数量计算应付金额。

④ 增添新的书名、数量和单价。

⑤ 当某种书数量为0时,将它自动删除。

第九章 联合和枚举

联合和枚举是 C 语言中两种构造类型。这两种类型比数组和结构应用较少。联合与结构很相似,但其区别在于结构是异址的,而联合是同址的。枚举在定义形式上与结构、联合有相似之处,但它却是另一种类型,它是若干常量集合。

9.1 联合的概念

9.1.1 联合变量的定义和赋值

联合变量定义的形式与结构很相似,除关键字不同,其余几乎相同。下面是一个联合模式的定义格式:

```
union <联合名>
{
    <联合成员说明>;
};
```

其中,union 是联合的关键字。<联合名>的命名方法同标识符,<联合成员说明>是将组成该联合的所有成员进行类型说明。C 语言中,几乎所有类型都作为联合的成员,包含结构变量在内。

联合变量的定义如下所示:

```
union <联合名> <联合变量名表>;
```

定义一个联合变量之前要先定义一种联合模式,任何一种联合变量都是属于某种联合模式的联合变量,这一点与结构变量很相似。<联合变量名表>中,可以是一个联合变量名,也可以是多个联合变量名,多个变量用逗号分隔。这里的变量可以是一般联合变量,也可以是指向联合变量的指针。例如:

```
union data
{
    char c_data;
    int i_data;
    float f_data;
    double d_data;
};

union data d1, d2, *pd;
```

其中,union 是关键字,data 是联合名,该联合有4个成员,分别作了说明。d1和 d2是两个联合变量,pd 是一个指向联合变量的指针。联合变量 d1和 d2都是具有联合名 data 的联合变量。而联合变量指针 pd 也是指向具有 data 联合名的联合变量的指针。

与结构相类似,上述定义联合变量也可以写成如下形式:


```

union data
{
    char c_data;
    int i_data;
    float f_data;
    double d_data;
} d1, d2, *pd;

```

这里,也同样定义了联合变量 d1和 d2以及指向联合变量的指针 pd。

联合变量的成员表示也与结构相似。联合变量的成员用“.”表示,指向联合变量的指针用“->”表示。

联合变量的赋值主要是给联合变量的各成员赋值。例如,在上例中,对联合变量 d1的几个成员赋值分别如下:

```

d1.c_data='a';
d1.i_data=15;
d1.f_data=10.5;
d1.d_data=87.63;

```

对指向联合变量的指针 pd 赋值与给联合变量赋值相同,只是其成员表示不同。例如,给 pd 的最后一个成员赋值如下:

```
pd->d_data=97.0;
```

由于联合变量中的若干个成员共用内存单元,即一个联合变量的所有成员具有一个相同的内存地址值,因此,在联合变量中起作用的成员是最近一次被赋值的成员,因为一个联合变量的若干个成员共用一个内存地址,存入了新的成员值时,原来的成员值便失去意义。

如果给联合变量赋初值,只能有一个值,并且指定赋给第一个成员,而不能用多个值赋给多个成员,其原因是由于联合变量所有成员共占一个内存地址。

9.1.2 联合与结构的区别

前面对联合变量的定义和联合变量成员的表示以及对联合变量的赋值作了描述,可以看出联合与结构有许多相似之处,但是也指出了两者的最大区别在于联合是共址的,结构是异址的。即联合的所有成员共同使用一个内存地址,而结构的每个成员都有自己的内存地址。

由于联合的共址特性使得它与结构产生了很大差别。例如,在赋初值时,联合变量只能给第一个成员赋初值;不能对联合变量名赋值;不能用联合变量作函数参数,也不能用联合变量作函数的返回值,只能用指向联合变量的指针作函数参数。

下面举出两个例子说明联合变量的使用和特点。

[例9.1] 分析下列程序的输出结果,进而说明联合变量的成员是共址的。

```

union data
{
    char c_data;
    int i_data;
    float f_data;
};
main()

```

```

{
    union data d1;
    d1.c_data='a';
    d1.i_data=5;
    d1.f_data=3.7;
    printf("%c\t%d\t%.2f\n",d1.c_data,d1.i_data,d1.f_data);
    printf("%d\n",sizeof(d1));
    printf("%p\t%p\t%p\t%p\n",&d1.c_data,&d1.i_data,&d1.f_data,&d1);
}

```

执行该程序输出结果如下:

```

?    ?    3.7    (?表示无意义)
4
FFD6    FFD6    FFD6    FFD6

```

说明:

(1) 该程序中,首先定义一个联合,其名为 data,它有3个成员,分别是三种不同类型。又定义联合变量 d1,并给它的三个成员分别赋值。当使用 printf() 函数输出 d1 的三个成员的值时,前两个成员输出值是无意义的,只有最后一个成员是有意义的,其值为 3.7。这说明:某一时刻一个联合变量中只有一个成员起作用,其他成员不起作用。

(2) 输出 sizeof(d1) 的值为 4,这说明联合变量 d1 占内存 4 个字节。在多个联合成员共占一个内存地址时,该地址所指向的内存空间是所有成员中占内存空间最大的成员所占的内存空间。该例中的三个成员所占内存字节数分别为 1,2 和 4,最大的是 4,因此,联合变量 d1 所占内存空间为 4 个字节。

(3) 使用 printf() 函数分别输出联合变量 d1 的三个成员的内存地址都是相同的,并且与联合变量 d1 的地址值也是相同的,可见联合变量各成员是共址的。

[例9.2] 分析下列程序的输出结果,并指出该结果说明了什么问题。

```

main()
{
    union {
        int ig[6];
        char s[12];
    }try;
    try.ig[0]=0x4542;
    try.ig[1]=0x2049;
    try.ig[2]=0x494a;
    try.ig[3]=0x474e;
    try.ig[4]=0x0a21;
    try.ig[5]=0x0000;
    printf("%s\n",try.s);
}

```

执行该程序输出如下结果:

```
BEIJ JING!
```

说明: 该程序中定义一个无名联合,用它定义一个联合变量 try,该联合有 2 个成员,每个

成员都占内存12个字节。程序中对 try 的 ig 成员赋了值,ig 是一个 int 型数组,分别对它的6个元素都赋了值。然后,程序中通过 try 的另一个成员 s 进行输出,所输出的字符串正是 try 的 ig 成员所被赋值的 ASCII 码所对应的字符组成的。由此可见,联合变量各个成员是共占内存单元的,因此,按某个成员赋的值,可按其另一个成员进行输出,但要求输出的类型与数据类型相一致。

9.2 联合的应用

由联合的特征决定了它的应用远不如结构应用那样广泛。但是,在有些情况下也使用联合来解决问题。在实际应用中,常常会出现一些量相互间排斥的情况,这时用联合就十分方便。例如,假定某学校一些学生在校内住宿,另一些学生在校外住宿。对于在校内和在校外住宿的学生的住址描述是不同的,可分别用下述两种结构来描述:

```
struct off_school
{
    int strnum;
    char strname[20];
    char city[20];
};
struct in_school
{
    char collname[10];
    char dorm[10];
    int roomnum;
};
```

住在校外的学生可用 off_school 结构来描述地址,而住在校内的学生可用 in_school 结构来描述地址。而对每个学生来说,情况是唯一的,即二者只可选择其一。于是,一个学生的地址可用下述联合来描述:

```
union address
{
    struct off_school town;
    struct in_school gown;
};
```

该联合中两个成员是结构变量,即结构变量可以作联合成员;相反,联合变量也可以作为结构成员,即联合与结构二者可以相互嵌套。例如:

```
struct student
{
    char name[20];
    int stnum;
    int grade[3];
    union address a;
};
```

该结构中,有一个成员 a 是联合变量。而该联合中又有结构变量。这便是结构和联合的定义上的嵌套。

虽然,由于联合成员是共址的,而对联合变量应用作了一些限制,不像结构变量应用得那么广泛,但是联合变量除了可以作为结构成员外,还可以作为数组元素,即联合数组。另外,指向联合变量的指针可以作函数参数等。

[例9.3] 假定描述一个学生使用下面的内容:学号,姓名,三门功课的成绩和住址。其中,住址有两种情况:住在校内和住在校外。使用一个变量来标识一个学生住在校内或是校外,而学生住址用前面讲过的联合 address 来表示。于是描述学生的结构格式如下:

```
struct student
{
    int stunum;
    char name[20];
    int grade[3];
    char off_in;
    union address a;
};
```

其中,char 型变量 off_in 用来标识该学生是住在校内(用 'n' 表示)还是住在校外(用 'f' 表示)。

编程输入每个学生的信息,并通过姓名来查找某个学生的住址和三门成绩总和。

程序内容如下:

```
struct off_school
{
    int strnum;
    char strname[20];
    char city[20];
};
struct in_school
{
    char collname[10];
    char dorm[10];
    int roomnum;
};
union address
{
    struct off_school town;
    struct in_school gown;
};
struct student
{
    int stunum;
    char name[20];
    int grade[3];
    char off_in;
    union address a;
```

```

} s[3] = {{7001, "Li", {90, 80, 85}, 'f'},
          {7002, "Ma", {85, 95, 87}, 'f'},
          {7003, "Lu", {80, 75, 83}, 'n'}};

main()
{
    int i;
    char name[20];
    for(i=0; i<3; i++)
    {
        printf("Input address -- ");
        if(s[i].off_in == 'f')
        {
            printf("strnum, strname, city: ");
            scanf("%d%s%s", &s[i].a.town.strnum, s[i].a.town.strname,
                  s[i].a.town.city);
        }
        else
        {
            printf("collname, dorm, roomnum: ");
            scanf("%s%s%d", s[i].a.gown.collname, s[i].a.gown.dorm,
                  &s[i].a.gown.roomnum);
        }
    }
    printf("Input name: ");
    scanf("%s", name);
    for(i=0; i<3; i++)
    {
        if(!strcmp(s[i].name, name))
        {
            if(s[i].off_in == 'f')
            {
                printf("%d, %s, %s\n", s[i].a.town.strnum, s[i].a.town.strname,
                        s[i].a.town.city);
                printf("%d\n", s[i].grade[0]+s[i].grade[1]+s[i].grade[2]);
            }
            else
            {
                printf("%s, %s, %d\n", s[i].a.gown.collname, s[i].a.gown.dorm,
                        s[i].a.gown.roomnum);
                printf("%d\n", s[i].grade[0]+s[i].grade[1]+s[i].grade[2]);
            }
        }
    }
}

```

执行该程序，屏幕上显示如下信息：

```

Input address--strnum, strname, city: 101 Haidian Beijing ✓
Input address--strnum, strname, city: 203 Haidian Beijing ✓
Input address--collnum, dorm, roomnum: Beida 35d 105 ✓
Input name: Ma ✓

```

这时,屏幕上显示该程序的输出结果:

203, Haidian, Beijing

267

说明:该程序是结构和联合相互嵌套,结构中有联合变量,联合中又有结构变量。程序中定义的s是结构数组,它有三个元素。开始时对每个元素的前4个成员赋了初值,然后在程序中又通过键盘输入给s的3个元素中最后一个成员赋值,由于该成员的值有两种形式,或者是校外地址形式,或者是校内地址形式,因此使用了下列if语句:

```
if (s[i]. off_in == 'f')
    :
```

满足该if条件,则按校外地址格式输入地址,否则按校内地址格式输入地址。同样,在输出学生地址时,也有两种不同形式。

9.3 枚举的概念

9.3.1 枚举变量的定义和赋值

枚举也是一种构造的数据类型,具有这种类型的变量,称为枚举变量。枚举变量的定义形式与结构变量、联合变量有相似之处,但是枚举变量与它们都有很大的不同。

枚举是具有名字的若干个常量的有序集合,枚举变量的取值范围是该枚举表所对应的枚举符。枚举变量被赋值以后,它实际上是一个常量,因为枚举符是具有名字的常量,而枚举变量的值只能取某一个枚举符。因此,有人说,枚举变量是一种特殊的常量。

枚举变量在定义之前要先定义一种枚举模式,任何一枚举变量都是某种枚举模式的枚举变量。枚举模式定义格式如下:

```
enum <枚举名>{<枚举表>;
```

其中,enum是枚举关键字,<枚举名>的命名方法同标识符。<枚举表>是由若干个枚举符组成的,多个枚举符之间用逗号(,)分隔。枚举符又称为枚举元素或枚举常量,它是一种标识符,而且它具有确定的int型值。

枚举变量定义格式如下:

```
enum <枚举名> <枚举变量名表>;
```

其中,<枚举变量名表>是由逗号分隔的若干个枚举变量名组成的。例如:

```
enum day{Sun, Mon, Tue, Wed, Thu, Fri, Sat};
enum day d1, d2;
```

其中,day是枚举名,Sun, Mon, ..., Sat是枚举符表,它由7个枚举符组成。d1和d2是被定义的具有day枚举模式的两个枚举变量。d1和d2的值只能选取day的枚举符。

另外,d1和d2的定义也可以写成如下格式:

```
enum day{Sun, Mon, Tue, Wed, Thu, Fri, Sat} d1, d2;
```

定义后枚举变量应该先赋值,然后再引用,否则无意义。枚举变量应被赋一个它所对应的枚举符表中的一个枚举符。例如:

```
d1=Sun;
```

```
d2=Fri;
```

这时,d1和d2两个枚举变量被赋了值,而Sun和Fri都是d1和d2所对应的枚举模式的枚举表中的枚举符,因此这是合法的。这里必须指出一点,不能直接给枚举变量赋一个整型数值,例如,

```
d1=0;
```

这是非法的,而

```
d1=(enum day) 0;
```

是合法的,它等价于

```
d1=Sun;
```

因为,Sun枚举符所隐含的int型数值为0。在给枚举变量赋int型数值时,前面必须加上强制类型运算符(enum(枚举名))。

枚举表中的枚举符不是变量,而是具有名字的常量,它们都各自隐含一个int型值。在默认的情况下,枚举表中枚举符的值从0开始,后一个总是比前面一个大1。例如,在上述的枚举表中,Sun的值是0,Mon的值是1,Tue是2,Wed是3,Thu是4,Fri是5,Sat是6。另外,在定义枚举模式时,可以通过显式赋值的方法来确定枚举符的值。例如:

```
enum day {Sun=7, Mon=1, Tue, Wed, Thu, Fri, Sat};
```

这里,Sun的值是7,Mon的值是1,它们是通过显式赋值来确定其值的,Tue没有被显式赋值,它的值是前一个枚举符的值加1,即为2,同样Wed的值是3,...,Sat的值是6。

下面再举一个枚举的例子。

```
enum coin {penny, nickel, dime, quarter, half_dollar, dollar}money;
```

该枚举名是coin,它由6个枚举符组成一个枚举表,money是枚举变量名,它的取值范围在该枚举表中。例如,

```
money=dime;
```

是合法的,而

```
money=Fri;
```

是非法的,因为Fri不是money枚举变量所对应的枚举表的枚举符,而是day枚举名的枚举符,因此,任何一个枚举变量的值只能是它所对应的枚举表中的枚举符,而不能是其他枚举表中的枚举符。

9.3.2 使用枚举变量时应注意的事项

在编程中使用枚举变量应注意如下事项。

(1) 枚举符不是整型变量,不能在程序中对它赋以数值。在上述day枚举例中,例如,

```
Mon=2;
```

```
Sat=5;
```

都是错误的。

枚举符是按常量处理的,称为枚举常量。枚举符所隐含的int型值可以在定义枚举模式时对枚举表中的枚举符进行显式赋值来确定。

(2) 枚举变量一般用它所对应的枚举表中的枚举符来赋值。如果用整型值来赋值时,前面

需要加上强制类型运算符,而不能直接用 int 值来赋值。枚举变量的输出值是 int 型值,而不是字符串。枚举表中的枚举符只是一个有名字的 int 型值,把它赋给枚举变量,而使枚举变量获得了其名字所隐含的 int 型值。因此,输出枚举变量值时,要使用格式符 %d。例如,d1 是枚举名 day 的一个枚举变量,而被赋值为 Wed,输出该变量值使用如下格式:

```
printf("%d\n", d1);
```

如果你希望输出 Wed 时,还需要做一些转换,其方法很多,可以用字符数组的方法,如本章后面的例子,也可以用 switch 语句的方法,读者可自行设计。

(3) 枚举变量可以进行比较运算。比较时按其枚举符所隐含的枚举值进行。例如,

```
if(d1 == Tue)...
```

```
if(d1 < Sat)...
```

都是合法的。

另外,枚举变量可以用作函数参数和函数的返回值。有关例子在后面章节中会看到。

9.4 枚举的应用

使用枚举变量的好处主要是更加直观,用户可以选用一些“见名知意”的枚举符,使人看上去一目了然,使用枚举符可便于记忆。另外,枚举变量的值受到所对应的枚举表的限制,增加数据的安全性,一旦枚举变量所得到的值超过了相应的枚举表的范围,则会出现错误信息。枚举类型的数据在编译时作类型检查,增加了可靠性。

枚举变量的主要用途是可作为函数参数和函数返回值。枚举变量也可以作结构的成员。

[例9.4] 编一个程序,已知某天是星期几,计算出下一天是星期几。要求使用枚举变量。程序内容如下:

```
enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
enum day day_after(d)
enum d;
{
    return((enum day)(((int)d+1)%7));
}
main()
{
    enum day d1, d2;
    static char *name[] = {"Sun", "Mon", "Tue",
                           "Wed", "Thu", "Fri", "Sat"};
    d1 = Sat;
    d2 = day_after(d1);
    printf("%s\n", name[(int)d2]);
}
```

执行该程序输出结果如下:

Sun

说明:

(1) 本例中,被调用函数的参数使用了枚举变量,并且该函数的返回值也是枚举变量。因

此,可见枚举变量可用来作函数参数和函数的返回值。

(2) 枚举符是一个名字,它具有 int 型值。可用它直接给枚举变量赋值,但要求是对应枚举表中的枚举符。但是,不能使用 %s 格式直接输出枚举符的名字。如果需要输出枚举符的名字时,该例是通过字符数组进行转换的。

[例9.5] 已知一口袋中有红、白、黄、蓝、黑五种球若干个。每次取3个球,编程计算每次取出三种不同颜色的球的所有可能的组合,并打印出这些不同的组合。

程序内容如下:

```
enum color {black,blue,red,yellow,white};
main()
{
    int n=0,m;
    enum color i,j,k,p;
    for(i=black;i<=white;i++)
        for(j=black;j<=white;j++)
            if(i!=j)
                for(k=black;k<=white;k++)
                    if((k!=i)&&(k!=j))
                    {
                        n++;
                        printf("%5d",n);
                        for(m=1;m<=3;m++)
                        {
                            switch(m)
                            {
                                case 1:p=i; break;
                                case 2:p=j; break;
                                case 3:p=k;
                            }
                            switch(p)
                            {
                                case black: printf("%-10s","black");
                                            break;
                                case blue: printf("%-10s","blue");
                                            break;
                                case red: printf("%-10s","red");
                                           break;
                                case yellow: printf("%-10s","yellow");
                                           break;
                                case white: printf("%-10s","white");
                                           break;
                            }
                            printf("\t");
                        }
                        printf("\n");
                    }
    printf("\n");
}
```

执行该程序输出结果读者自己分析。

说明：该程序中定义了枚举变量，但这种定义并不是必须的。由于定义了枚举变量 i, j, k 和 p, 使得它们的取值范围有了限制，增加了安全性，即 i, j, k 和 p 取值范围超出了 color 的枚举表中的枚举符，将出现错误信息；同时，在编译时对这4个变量将进行类型检查，这样又增加了可靠性。

本例将通过使用 switch 语句，将枚举变量 p 的值转换为枚举符进行输出，这样更加直观一些。

练习 题

1. 什么是联合？如何定义联合变量？
2. 联合变量的成员如何表示？如何对联合变量赋值和赋初值？
3. 比较联合与结构有哪些异同点？指出联合与结构的本质区别是什么？这一区别带来了什么影响？
4. 联合变量有哪些应用？
5. 什么叫枚举？枚举变量如何定义？
6. 枚举变量有什么特点？在使用枚举变量时应注意什么事项？
7. 枚举变量有哪些应用？

作 业 题

1. 判断下列描述是否正确，对者划√，错者划×。
 - (1) 联合变量的定义与结构变量的定义除其关键字不同之外，其余很类似。
 - (2) 联合变量的成员表示与结构变量的成员表示是一样，都使用运算符·或->。
 - (3) 联合变量可以作为结构成员，结构变量也可以作为联合成员，两者是可以相互嵌套的。
 - (4) 联合变量可以像结构变量一样的赋初值，使用其初始值表。
 - (5) 联合变量与结构变量的最大区别是联合变量的成员是同址，而结构变量的成员是异址的。
 - (6) 可以将一个已具有数值的联合变量赋给另一个相同联合模式的联合变量。
 - (7) 枚举是具有名字的若干个常量的集合，枚举变量的值只能取这一集合中的某个枚举符。
 - (8) 枚举定义的枚举表中的枚举符必须都要显式地指定其整数值。
 - (9) 枚举变量可以将其对应的枚举表中的枚举符赋给它，也可以将一个整数直接赋给它。
 - (10) 枚举变量可以作为函数的参数和函数的返回值。
2. 选择填空。
 - (1) 枚举符的默认的整数值是从()开始，而后边一个没有被显式赋值时其值为前一个加()。
A. 0 1 B. 1 1 C. 1 0 D. 0 0
 - (2) 联合变量成员的内存地址是()，所占的内存字节数是()。
A. 相同的 相同的 B. 相同的 不同的
C. 不同的 相同的 D. 不同的 不同的
 - (3) 枚举符是()。
A. 具有整型数值的字符串 B. 具有名字的整型值
C. 一个以名字表示的字符串 D. 一个必须显示赋值的整型数
 - (4) 联合变量()作函数参数，指向联合变量的指针()作函数参数。

A. 不能 不能

B. 不能 可以

C. 可以 不能

D. 可以 可以

(5) 已知 int 型变量占2个字节, float 型变量占4个字节, char 型变量占1个字节, 下列定义的结构变量 a 占 () 字节。

```
union x
{
    int i;
    float j;
};
struct y
{
    char c[5];
    union x d;
}; a
```

A. 9 B. 10 C. 8 D. 12

(6) 写出下列程序段中, y.n 值按 %x 输出为 ()。已知, 'a' 的 ASCII 码值为 97。将两个元素的字符数组放在一个 int 型数组中(16位机), 并假定先放低位, 再放高位。

```
union x
{
    char m[2];
    int n;
} y;
y.m[0]='c'; y.m[1]='e';
```

A. 6563 B. 6365 C. 3656 D. 5636

(7) 在下列枚举定义中, 枚举符 D 所表示的整型数值是 ()。

```
enum M { A=8, B=4, C, D, E=1};
```

A. 5 B. 6 C. 0 D. 3

(8) 下列定义的错误是 ()。

```
union a
{
    int a;
    float b;
    char a;
} a;
```

A. 联合名与联合变量同名

B. 联合名与联合变量的成员同名

C. 联合的两个成员同名

D. 联合名, 联合变量名与联合成员同名

3. 分析下列程序的输出结果。

(1)

```
main()
{
    union {
        int di;
        char dc;
        float df;
        double dd;
```

```

        char *pd;
    }d1;
    d1.dc='a';
    printf("d1 size-->%d\n",sizeof(d1));
    printf("d1.dd size-->%d\n",sizeof(d1.dd));
    printf("d1.di:%d\n",0x00ff&d1.di);
}

(2)
struct tag
{
    char low,high;
};
union word
{
    struct tag byte;
    short word;
}w;
main()
{
    w.word=0x6b7d;
    printf("word value: %x\n",w.word);
    printf("low value: %x\n",w.byte.low);
    printf("high value: %x\n",w.byte.high);
    w.byte.high=0x56;
    printf("word value: %x\n",w.word);
}

(3)
enum coin {penny,nicel,dime,quarter,half_dollar,dollar};
char * name[]={ "penny", "nicel", "dime", "quarter",
                "half_dollar", "dollar" };
main()
{
    enum coin money1,money2;
    money1=dime;
    money2=dollar;
    printf("%d,%d\n",money1,money2);
    printf("%s,%s\n",name[(int)money1],name[(int)money2]);
}

```

4. 使用结构和联合及枚举编写程序。

现有铅笔和圆珠笔若干支。铅笔的数据包括：品名(用铅笔)、数量、价格和类别，其中类别包含0.3的、0.5的和0.7的三种；圆珠笔的数据包括：品名(用圆珠笔)、数量、价格和类别，其中类别包含：红色的、蓝色的、黑色的。要求：铅笔和圆珠笔使用一种结构模式来描述，其中类别成员联合类型，三种不同的类别的选取用枚举。程序要求实现：

- (1) 输入若干支不同类型的铅笔和圆珠笔。
- (2) 统计输出某一种或二种笔的总价值(即支数乘以单价)。

第十章 文件和读写函数

本章讲述 C 语言中文件的概念,讲述标准文件和一般文件的读写函数以及常用的库函数。

10.1 C 语言中文件的概念

本节着重讲述什么是文件,什么是文件指针,一般文件和标准文件的区别等有关文件的基本概念。

10.1.1 文件和文件指针

1. 文件

一般说来,文件是有序数据的集合。程序文件是程序代码的有序集合,数据文件是一组数据的有序集合。文件是被存放在外部存储设备中的信息。对文件的处理过程就是面向文件的输入和输出过程。文件的输入过程是从文件中读出信息,文件的输出过程是往文件中写入信息,文件的输入的过程使用读函数,实现文件输出的过程使用写函数。文件的读写函数是实现文件操作的主要函数,本章将用大量篇幅来讲述文件的读写函数。

C 语言文件被称为流式文件,其特点是不分记录或块,将文件看成是信息“流”或看成是一个字符流(文本文件),或看成是一个二进制流(二进制文件)。文件的存取是以字符(字节)为单位的,读写数据流的开始和结束受程序控制。任何一个文件都是以 EOF 结束,最简单的文件是只有结束符的空文件。

C 语言文件包含有设备文件和磁盘文件,例如,键盘是一种输入信息的文件,显示器屏幕和打印机是输出信息的文件它们都属于设备文件。将内存的信息放到磁盘上保存,需要时再从磁盘上装入内存,这就要使用磁盘文件,磁盘文件是计算机中常用的文件。

C 语言文件按存放设备分设备文件和磁盘文件;按数据的组织形式分为文本文件(ASCII 码文件)和二进制文件。文本文件是按一个字节存放一个字符的 ASCII 码来存放的;二进制文件是按数据在内存中的存储形式放到磁盘上的。例如,有一个整数 10000,在内存中按二进制形式存放,占 2 个字节,将它放在磁盘上如按文本文件形式存放,占 5 个字节,每个数位占一个字节。两种存放方式各有利弊。以文本文件形式输出便于对字符进行处理,也便于输出字符,但是占用存储空间较多,并且要花费转换时间。以二进制文件形式输出可节省存储空间和转换时间,但是不能直接输出字符形式。

2. 文件指针

文件指针是一种用来指向某个文件的指针。如果说某个文件指针指向某个文件,则是该文件指针指向某个文件存放在内存中的缓冲区的首地址。

每一个被使用的文件都要在内存中开辟一个区域,用来存放的有关信息,包括文件名字、文件状态和文件当前位置等。这些信息被保存在一个结构变量中,该结构变量所对应结构模式

被系统定义为 FILE, 它被放在 stdio.h 文件中。有些版式的 FILE 被定义如下:

```
type struct
{
    int    fd;          /* 文件号 */
    int    cleft;       /* 缓冲区内剩余的字符 */
    int    mode;        /* 文件操作模式 */
    char *  mnextc;     /* 下一个字符位置 */
    char *  buff;       /* 文件缓冲区位置 */
}FILE;
```

在文件操作的程序中, 要使用 FILE 来定义文件指针, 并且将打开的文件缓冲区的首地址赋给文件指针, 让它指向该文件。例如,

```
FILE * fp;
```

其中, fp 是一个指向文件的指针。

```
fp=fopen("abc.txt", "r");
```

给 fp 赋值, 使它指向 abc.txt 文件。于是, fp 便是一个指向 abc.txt 文件的指针。有了文件指针以后, 对文件的操作(读、写和关闭等)都使用文件指针, 而不使用文件名。

3. 读写指针

读写指针是指当一个文件被打开后用来标识读写文件位置的。它与文件指针是不同的。文件指针一旦被指向某个文件, 它的值是不会改变的, 直到该文件被关闭为止。而读写指针是当某个文件被打开时, 它指向文件头或文件尾(与打开方式有关), 可以通过定位函数(fseek())来改变读写指针的位置。可见, 读写指针与文件指针是两个完全不同的概念, 在使用时应注意分清。

关于读写指针的详细描述, 在本章后面讲述文件定位函数时还会讲到。

10.1.2 标准文件和一般文件

1. 标准文件

C 语言中规定的标准文件有三个, 它们分别是标准输入文件(键盘)、标准输出文件(显示屏幕)和标准出错信息文件, 规定错误信息显示在屏幕上。这三个文件的文件指针分别为标准输入文件是 stdin, 标准输出文件是 stdout, 标准出错信息文件是 stderr。

标准文件的特点是这类文件使用前不必打开, 使用后不必关闭。因为系统将它启动系统时自动打开, 在退出系统时自动关闭, 并且自动为这三个标准文件分配缓冲区, 指定文件指针。因此, 使用标准文件十分方便。这也是在前面所讲述的内容中没有涉及到文件打开和关闭操作的原因。到现在为止, 所使用的读写函数(即输入输出函数)都是对标准文件的, 而对于一般文件(即非标准文件)的操作在本章后面再讲述。

2. 一般文件

一般文件是指除了上述的标准文件以外的文件, 包括设备文件和磁盘文件。

一般文件的特点是操作前需要先打开文件, 操作后要及时关闭文件。打开文件和关闭文件由专门的函数实现这一操作。执行打开文件函数实现打开文件的操作就是在内存中建立一个存放文件的缓冲区。如果打开文件成功, 则内存建立了一个缓冲区, 这时打开文件函数将返回一个地址值, 将它赋给一个定义的文件指针, 让它指向该文件。如果打开文件失败, 则内存中不

建立缓冲区,这时打开文件函数返回 NULL。一旦文件被打开后,便可以对文件进行读或写操作,对于一般文件来讲,打开文件是进行读写操作的前提。打开的文件操作完成后,要及时关闭文件,关闭文件由专门关闭文件函数来实现。及时关闭文件可以及时释放所占用的内存空间,还可以保证文件内容的安全。关闭文件是将文件从内存中清除,送回到磁盘中,因此,不要把关闭文件看成是删除文件。应该养成及时关闭不用文件的好习惯。

10.1.3 高级读写函数和低级读写函数

UNIX 系统下的 C 语言版本对文件的处理方法分成两种:一种叫“缓冲文件系统”,另一种叫“非缓冲文件系统”。缓冲文件系统是指系统在内存区域中自动地为打开的文件开辟一个缓冲区,对文件数据的读写都要经过缓冲区。具体操作是当从内存中输出数据时,先将数据送到内存缓冲区,装满缓冲区后一起送到磁盘;当从磁盘向内存装入数据时,则是从磁盘文件中一次将一批数据送到内存装满缓冲区,然后再从缓冲区逐个地将数据送到程序数据区,赋给程序变量。一般地,缓冲区大小为512个字节。非缓冲区是指系统不为文件自动建立缓冲区,而是由程序为每个文件设定缓冲区。

一般地认为,使用缓冲文件系统进行文件读写操作的称为高级读写函数,它与机器无关;使用非缓冲文件系统进行文件读写操作的称为低级读写函数,它与机器相关。早先规定用缓冲文件系统处理文本文件,用非缓冲文件系统处理二进制文件,后来将缓冲文件系统扩充为可以处理二进制文件。

本书着重讲解高级读写函数的使用,考虑到低级读写函数使用较少,本书将不再介绍,如果需要可阅读有关资料。

C 语言中,设有输入输出语句,而对文件的读写操作的函数都是用库函数来提供的。

10.2 标准文件的读写操作

前面已经介绍过了标准文件的特点。对标准文件实行操作,不需要打开和关闭文件的操作,因为系统对标准文件自动打开和关闭,只需要读写操作。

10.2.1 标准文件读写函数介绍

在本书的第一章中对标准文件的读写函数已经作了较为详细的讲述。这里,再作一些简单介绍和补充。

1. 对一个字符的读写函数

读一个字符的函数 `getchar()`,该函数用来从键盘缓冲区中每次读取一个字符,该函数的返回值为读取字符的 ASCII 码值。

写一个字符的函数 `putchar()`,该函数有一个参数,其功能是将参数给出的字符输出到屏幕上。

2. 对一个字符串的读写函数

读一个字符串函数 `gets()`,该函数返回一个字符型指针,该函数有一个参数,该参数是用来存放读取的字符的。该函数的功能是从键盘上读取一个字符串存放到该函数参数所指定的字符数组中,返回指向该字符数组的指针。

写一个字符串函数 puts(), 该函数的功能是将其参数中所指定的字符串输出到显示屏幕上。该函数的参数是一个字符数组或字符指针, 或字符串常量。

3. 具有指定格式的读写函数

这里再补充介绍标准格式输入函数 scanf() 和标准格式输出函数 printf() 的一些内容。

(1) 标准格式输入函数 scanf()

该函数格式如下:

```
scanf("<控制串>", "<参数表>")
```

该函数有关参数说明在第一章中已经讲述过了。下面仅作两点补充。

一是在<控制串>中除了格式符外, 还可用除空格符以外的一般字符, 一般字符作为匹配符。所谓匹配符是用来确定输入数据流中输入项的。在控制串中使用一般字符作为匹配符时, 输入数据流中要有与匹配符对应的字符来区分数据流中的输入项。例如, 控制串中出现匹配符逗号(,), 则在输入流中要以逗号来分隔输入项, 要求控制串中出现的匹配符要与输入流中出现的分隔符一致, 即字符相同, 次数相等。

C 语言中用来区分输入流的数据输入项的方法有三: 一是前面讲述的用匹配符的办法; 二是在格式符中加最大域宽的修饰符; 三是使用默认的输入项分隔符——空白符, 这时控制串中不加任何其他字符。

二是在控制串中, 格式符里 % 与格式说明符之间可以加如下修饰符:

(1) 数字。用来表示最大的域宽。如果输入流中数据位数超过最大域宽, 则只取最大域宽的位数, 其后舍去; 如果输入流数据位数不足最大域宽, 则按实际宽度读取。

(2) l 字符。l 字符用在表示 int 型数的格式符 d, o, x 前面, 表示长整型; 用在格式符 f 前面表示双精度浮点数。

(3) h 字符。h 字符只可用在表示 int 型数的格式符 d, o, x 前面, 表示 short int 型。

(4) * 字符。这是一个抑制符, 表示跳过 * 后面整数所指出的输入域宽, 如果没有表示跳过的整数域宽, 则跳过一个输入域, 到下一个空白符。

[例10.1] 分析下列程序的输出结果。注意标准格式输入函数的使用。

```
main()
{
    int a,b,c;
    printf("Enter a b c: ");
    scanf("%2d%*3d%3d%4d",&a,&b,&c);
    printf("a=%d,b=%d,c=%d\n",a,b,c);
}
```

执行该程序输出如下信息:

Enter a b c: 1234567890 ✓

输出结果如下:

a=12, b=678, c=90

说明: 该例中 scanf() 函数中格式符里使用了数字来规定输入项的最大宽度, 并用星号符来抑制某些输入项。该控制串中各格式符的规定说明如下: 将前2位整数赋值给变量 a, 然后跳过3位整数, 再将3位整数读取给变量 b, 最后还剩下2位数读取给变量 c, 变量 c 对应的输入最

大宽度为4位,不足4位有几位取几位。在实际应用中,常用抑制符来跳过不想输入的数据项。

(2) 标准格式输出函数 printf()

该函数的格式如下:

`printf("<控制串>",<参数表>)`

该函数有关参数说明在第一章中已经讲述过了。下面再作两点补充。

一是在<控制串>的格式符中,可以在%与格式说明之间加修饰符。该修饰符主要是用来指定输出数据项的宽度。常用的修饰符如下所述:

(1) 数字. 数字。小数点前的数字表示输出数据项的最小域宽,当实际输出数据项的宽度小于最小域宽时,按最小域宽输出,在右对齐的格式下,前面补空格符;当实际输出数据项的宽度大于最小域宽时,按实际宽度输出。小数点后面的数字表示输出数据项的精度。对浮点来讲,表示小数的位数;对字符串来讲,表示字符串中字符的最多个数;对整数来讲,表示输出项的最多位数。

(2) 负号-。负号用来表示输出数据项中的字符左对齐,没有负号时为右对齐。一般情况下,左对齐,右边补空格符;右对齐,左边补空格符。

(3) l 字符。l 字符用于表示 int 型数的格式符 d,o,x 的前面,表示长整型;用于格式符 f 的前面,表示双精度浮点数。

(4) 0 字符。0 字符用于右对齐格式中替换数据项左边的空格符。

(5) * 字符。* 字符用来通过变量的值指定域宽的。* 字符要在输出数据项的参数表中对应一个表达式,该表达式的值为该输出项的宽度。

[例10.2] 分析下列程序的输出结果。注意 printf() 函数中可变域宽和补0的应用。

```
main()
{
    int i=5;
    float x=12.345678;
    printf("%*.*3f\n",i,x);
    printf("%010.*f\n",i,x);
    printf("%0*.*f\n",i+6,i,x);
}
```

执行该程序运行结果如下:

```
12.346
0012.34568
00012.34568
```

说明:

(1) 在控制串“%*.*3f\n”中,* 对应的表示域宽的变量为参数表中的 i,i 值为5,则控制串相当于“%5.*3f\n”,该格式要求小数取3位,该数值整数有2位,加1位小数点,共6位,大于指定的最小域宽5,于是按实际宽度输出,最后一位四舍五入。

(2) 在控制串“%010.*f\n”中,* 对应的表示小数位数的变量为参数表中的 i,即5。格式中指定最小域宽为10,并且数据前面补的空格符用0替换。输出数据有2位整数,5位小数,再加1位小数点,共8位,因此数据前还应有2个0。

(3) 在控制串“%0*.*f\n”中,第一个* 对应的表示最小域宽的变量为参数表中 i+6 表

达式,其值为11,第二个*对应的表示小数位数的变量为参数表中的i,即5。格式中又指定用0替换数据前填补的空格符。因此,输出结果为00012.34568。

二是 printf()函数也有一个返回值,该值是表示输出函数所输出的所有数据项所占的总宽度。

[例10.3] 分析下列程序输出结果。注意分析 printf()函数的返回值。

```
main()
{
    int a,b,c;
    a=100;
    b=256;
    c=printf("%05d,%05d\n",a,b);
    printf("%d\n",c);
}
```

执行该程序输出结果如下:

00100,00256

12

说明: 程序中变量c是用来存放 printf()函数的返回值的。该函数的实际输出宽度是a、b变量各5位,逗号1位,换行符1位,共12位。

10.2.2 标准文件的读写函数应用

标准文件的读写函数在C语言程序中有着广泛的应用,因此熟练地掌握这些函数是十分重要的。特别是 scanf()和 printf()函数应用更多,而它们还有一定的复杂性,在应用中更应特别注意。下面讲述这些函数在使用中应该注意的一些事项。

(1) 由于 getchar()函数是带有缓冲区的,使用时应注意缓冲区剩余字符对后面输入函数的影响。

[例10.4] 分析下列程序的输出结果。

```
#include <stdio.h>
main()
{
    int i,j,k;
    printf("Enter 1 character: ");
    i=getchar();
    printf("%c\n",i);
    printf("Enter 1 character: ");
    j=getchar();
    printf("Enter 1 character: ");
    k=getchar();
    printf("%c,%c\n",j,k);
}
```

执行该程序后,显示如下信息:

Enter 1 character: a ✓

键入字符a和回车键后,输出如下信息:

a

Enter 1 character: Enter 1 character: b ↵

键入字符 b 和回车键后,输出如下信息:

,b

说明: 该程序中使用 `getchar()` 函数从键盘中接收字符, 当在“Enter 1 character:”提示信息后输入 a ↵ 后, 则输出信息如上所示, 其原因是因为键入的实际是两个字符, a 字符和回车符。将 a 字符送给变量 i, 将回车符送给了变量 j, 因此第二个“Enter 1 character:”提示则不需要再输入字符, 于是出了第三个“Enter 1 character:”提示, 当键入 b ↵ 后, 输出结果告诉我们, j 变量中的字符的确是回车符, 因为在输出, b 字符之前空了一行, 而 k 变量的字符为 b。

为了实现每个提示信息都要输入一个字符送给相应的变量, 就需要及时消除缓冲区内多余的字符。例如, 键入 a ↵ 后, 缓冲区中有两个字符, a 是有用的, 回车符是多余, 于是要将多余的回车符消除, 否则它将影响后面的继续输入。其办法可用 `getchar()` 函数来吃掉一个字符。上述程序可作如下修改。

```
#include <stdio.h>
main()
{
    int i,j,k;
    printf("Enter 1 character: ");
    i=getchar();
    getchar();
    printf("%c\n",i);
    printf("Enter 1 character: ");
    j=getchar();
    getchar();
    printf("Enter 1 character: ");
    k=getchar();
    printf("%c,%c\n",j,k);
}
```

执行该程序输出如下信息:

Enter 1 character: a ↵

a

Enter 1 character: b ↵

Enter 1 character: c ↵

b,c

说明: 该程序中增加了两条 `getchar()` 语句, 其目的是为了消除缓冲区中剩余的无用字符, 使程序得以正常进行。该程序中, 由于缓冲区中只有一个多余字符, 因此使用一次 `getchar()` 函数, 如果缓冲区中有多个剩余字符又该怎么办呢? 请读者思考。

(2) 标准格式输入函数 `scanf()` 也是带缓冲区的, 它也存在有剩余字符的及时处理问题。

[例 10.5] 分析下列程序输出结果。注意由于 `scanf()` 函数是带缓冲区的所产生的影响。

```
main()
{
```

```

char s1[10],s2[10];
printf("Input a string: ");
scanf("%4s",s1);
printf("s1: %s\n",s1);
printf("Input a string: ");
scanf("%s",s2);
printf("s2: %s\n",s2);
}

```

执行该程序显示如下信息:

```

Input a string: abcdef ✓
s1: abcd
Input a string: s2:ef

```

说明: 出现上述输出结果是因为键入 abcdef 加回车键后, 字符数组 s1 接受 4 个字符 abcd, 并自动加上字符串结束符。这时字符串缓冲区中还有 ed 和换行符。程序中输出显示 s1 后, 屏幕出现下一个 "Input a string:" 提示信息。由于缓冲区还有剩余字符, 则不再需要键盘输入信息, 而自动将缓冲区剩余字符 ef 赋给字符数组 s2。出现 s2:ef 的结果。

为了及时清除缓冲区中的剩余字符, 可在下一次使用 scanf() 之前, 用下述方法清除缓冲区中的剩余字符:

```
while (getchar());
```

读者可以试试。

(3) 在 scanf() 函数中, %c 与 %s 这种格式的区别主要表现在如下两个方面: 一是字符格式时变量只接收一个字符; 而字符串格式时除接收规定的字符外, 还自动加 '\0'。二是字符格式时, 遇到字符包含空格符都作为字符输入; 而字符串格式时, 非空格符前边的前导空格被略去, 不作为字符输入, 而从非空格符开始进行输入。

[例10.6] 分析下列程序的输出结果。注意格式符 %c 与 %s 的区别。

```

main()
{
    char s[10]="1234567";
    printf("Input s:");
    scanf("%4c",s);
    printf("s: %s\n",s);
    printf("Input s: ");
    scanf("%4s",s);
    printf("s: %s\n",s);
}

```

执行该程序输入如下信息:

```

Input s: LLLab ✓
s: LLLab567
Input s: LLLab ✓
s:ab

```

说明: 该程序中输入两个相同的字符串, 但是输出结果却不同, 其原因在于 scanf() 函数

中两次所用的格式符不同,一次用%4c,另一次使用%4s。于是,该程序的输出结果将十分清楚告诉我们,%c和%s这两种格式的区别,详细情况请读者自行分析。

(4) scanf()函数在正常情况下,它的参数表中的每个参数都能获得指定的数值,但在非正常情况,当某个参数所赋予的值与其指定的格式相矛盾时,则结束整个的输入,scanf()函数中后面的参数将没有被赋值。

[例10.7] 分析下列程序的输出结果。注意 scanf()函数的返回值和各变量的输出结果。

```
main()
{
    int a,n;
    char b;
    float c;
    n=scanf("%3d%c%f",&a,&b,&c);
    printf("%d\n",n);
    printf("%d,%c,%f\n",a,b,c);
}
```

当输入信息流如下所示:

123l l a l l 46.78 ✓

这时的输出结果如何呢?有人认为输出结果应该如下所示:

3

123,a,46.78000

可见,实际上的输出结果并非如此,为什么呢?请读者自己分析,并上机验证。

如果将输入信息流改为下述情况:

123a46.78 ✓

结果又将会怎样呢?进一步加深对%c格式的认识。

(5) 在实际中字符串的输入和输出可使用 gets()和 puts()函数比较方便。

[例10.8] 分析下列程序的输出结果。注意 gets()和 puts()函数的使用方法。

```
main()
{
    char s1[10],s2[10];
    printf("Input a string: ");
    gets(s1);
    puts(s1);
    printf("Input a string: ");
    gets(s2);
    puts(s2);
}
```

执行该程序显示如下信息:

Input a string: abcde ✓

abcde

Input a string: xyzmn ✓

xyzmn

说明：该程序中使用 `gets()` 函数从键盘上接收一个字符串，使用 `puts()` 函数将该字符串输出到屏幕上显示，使用起来十分方便。

实际中也可用 `gets()` 函数接收一个整型数，然后用 `atoi()` 函数将它转换成整型数。

10.3 一般文件的操作

一般文件操作包含文件的打开和关闭，文件内容的读写以及文件的定位等。

10.3.1 打开文件函数和关闭文件函数

一般文件在对它进行读写操作之前需要打开，打开文件函数格式如下：

`fp=fopen("〈文件名〉","〈打开方式〉")`

其中，`fopen` 是打开文件函数的名字，该函数有两个参数，一个是〈文件名〉，该参数是一个字符串，需用双撇号(″)括起，文件名要全名，即包含路径名和扩展名。另一个是〈打开方式〉，它也要用双撇号括起。打开方式的选择如下所示：

`r`：表示读方式
`w`：表示写方式
`a`：表示追加写方式
`r+w`：表示读写方式，也可写成 `r+`
`rb`：表示二进制文件读方式
`wb`：表示二进制文件写方式
`ab`：表示二进制文件追加写方式
`br+`：表示二进制文件读写方式

该函数如果成功地打开了指定的文件，则返回该文件的内存缓冲区的首地址，将它赋给一个被定义为文件指针的变量，于是该指针将指向这个被打开的文件，直到被关闭为止；如果该函数没有成功地打开文件，则返回值为 `NULL`。因此，在执行打开文件的操作后，总要判断一下返回值是否为 `NULL`，如果返回值为 `NULL`，则说明文件打开失败，一般应退出该程序，检查失败的原因。因此，常用如下程序段来对没有打开文件进行处理：

```
if(fp==NULL)
{
    printf("file can't open!\n");
    exit(1);
}
```

其中，`fp` 是文件指针，用它来接收 `fopen()` 函数的返回值。`exit()` 是一个退出当前的执行的程序的函数，当文件没有被打开时，一般应退出程序。

函数只有被成功的打开后才可以对它进行读写操作。一旦文件的读写操作完成后，应该及时地将它关闭。关闭文件函数的格式如下：

`fclose(fp)`

其中，`fclose` 是关闭文件函数的函数名，`fp` 是要关闭文件的文件指针。使用该函数可以将一个被打开的函数关闭，同时释放它所占用的内存缓冲区。当不使用该函数关闭打开的文件

时,当该程序执行完后,也会将打开的文件关闭。

关于打开文件函数 `fopen()` 和关闭文件函数 `fclose()` 将在后面举例说明。

10.3.2 一般文件读写函数及其使用

一般文件的读写函数共有4对,下面分别加以介绍,并且举例说明它的使用。

1. 对一个字符的读写函数

(1) 一个字符的读写函数 `fgetc()`

该函数的功能是从被打开的文件中读取一个字符。该函数的调用格式如下:

```
c=fgetc(fp);
```

其中,`fgetc` 是该函数的函数名,`fp` 是一个文件指针,`c` 是一个 `char` 型变量,使用 `fgetc()` 函数可从由 `fp` 指针所指向的文件中读取一个字符送给某个 `char` 型变量 `c`。反复使用该函数可以将某个文件从头到尾读一遍,直到文件结束,即遇到 EOF 时为止。如果 `fp` 为 `stdin` 时,该函数等价于 `getchar()` 函数。

使用该函数读取文本文件时,当读取到 EOF 时便可结束,因此要作如下判断:

```
(c=fgetc(fp))!=EOF
```

若满足上述条件,可继续读取;若不满足该条件,则结束读取。

使用该函数读取二进制文件时,常用 `feof()` 函数来判断是否文件结束,因为 EOF 被定义为 -1,用来判断二进制文件不太合适。`feof()` 函数调用格式如下:

```
feof(fp);
```

该函数返回值为1时表示文件结束,该函数返回值为0时表示文件没有结束。因此,在采用循环方式读取二进制文件内容时,循环语句的条件如下所示:

```
!feof(fp)
```

该表达式值为非0时,继续循环;该表达式值为0时表示文件结束,则停止循环。这种方法也可以用于文本文件的操作中。

(2) 一个字符的写函数 `fputc()`

该函数的功能是将一个指定的字符写到指定的文件中。调用格式如下:

```
fputc(c,fp);
```

其中,`fputc` 是该函数的函数名,`fp` 是一个文件指针名,`c` 是一个 `char` 型变量。该函数用来将变量 `c` 中的字符写到文件指针 `fp` 所指向的文件中。该函数成功时返回所写字符的代码值,失败时返回 EOF。

当 `fp` 为 `stdout` 时,`fputc(c,stdout)` 函数等价于 `putchar(c)` 函数。

[例10.9] 显示一个文件的内容。

```
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
    int c;
    FILE *fp;
    if(argc!=2)
```

```

{
    printf("Foemat:\n\t%s filename\n",argv[0]);
    exit(1);
}
if((fp=fopen(argv[1],"r"))==NULL)
{
    printf("Th file %s can't open.\n",argv[1]);
    exit(2);
}
while((c=fgetc(fp))!=EOF)
    putchar(c);
fclose(fp);
}

```

执行该程序时,将命令行参数中所指定的文件的内容显示在屏幕上。

说明:该程序主函数中使用了两个参数,argc 和 argv,它们分别用来存放命令行中参数的个数和参数的内容。执行该程序时,命令行需要有一个参数,该参数(实参)应该是被打开文件的文件名,即将该文件内容显示在屏幕上。假定该程序编译连接后可执行文件名为 typefile.exe,则命令行格式如下:

typefile fl.c

其中,fl.c 是要显示的文件的名称。执行该命令行后,fl.c 文件的内容被显示在屏幕上。

该程序是用 fgetc()函数,从 fp 文件指针所指向的文件中读取每一个字符,再用 putchar()函数将它显示在屏幕上。

[例10.10] 复制一个文件。

```

#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
    int c;
    FILE *fp,*fq;
    if(argc!=3)
    {
        printf("\007Usage:command file1 file2.\n");
        exit(1);
    }
    if((fp=fopen(argv[1],"r"))==NULL)
    {
        printf("\007The file %s can't open.\n",argv[1]);
        exit(2);
    }
    if((fq=fopen(argv[2],"w"))==NULL)
    {
        printf("\007The file %s can't open.\n",argv[2]);
        exit(3);
    }
}

```



```

while((c=fgetc(fp))!=EOF)
    fputc(c,fq);
fclose(fp);
fclose(fq);
}

```

执行该程序,则将命令行中参数1所指定的文件名中的内容复制到由参数2所指定的文件名中,假定该程序经过编译连接后,生成的可执行文件名为 copyfile.exe,则命令行格式如下:

```
copyfile f1.c f2.c
```

其中,f1.c 是源文件名,f2.c 是目标文件名,该程序将 f1.c 文件的内容复制到 f2.c 中,使得 f2.c 成为 f1.c 的一个副本。

该程序中,使用 fgetc()函数从 fp 文件指针所指向的文件中一个字符一个字符地读取,再使用 fputc()函数将读取的一个个字符写到由 fq 指针所指向的文件中。这里,fp 指向文件 f1.c,fq 指向文件 f2.c。

(3) 字符回送函数 ungetc()

该函数的功能是将预读的一个字符送回原文件中,即将文件的读写指针向文件头移动一个字符。该函数的调用格式如下:

```
ungetc(c,fp);
```

其中,ungetc 是该函数的名字,c 是被回送的字符,fp 是被操作的文件。该函数的使用请见下例。

[例10.11] 显示出已知文件 ab.txt 的第一个大写字母。已知文件 ab.txt 的内容如下所示:

```

abcdefGHJK
MNopq

```

程序内容如下:

```

#include <stdio.h>
#include <ctype.h>
main()
{
    int c;
    FILE *fp;
    fp=fopen("ab.txt","r");
    while((c=fgetc(fp))!=EOF)
        if (isupper(c))
            break;
    ungetc(c,fp);
    c=fgetc(fp);
    printf("%c\n",c);
    fclose(fp);
}

```

执行该程序输出结果如下:

```
G
```

说明:该程序中首先打开文件 ab.txt,然后使用 fgetc()函数从文件中逐一读取每个字

符,并用 `isupper()` 函数判断它是否是大写字母,该函数被包含在 `ctype.h` 文件中。如果读取的字符是大写字母,则停止对文件的继续读取,并使用 `ungetc()` 函数将读取的大写字母送回到文件中,实际上是将此时的文件读写指针向文件头方向移动一个字符,即使指向 'G' 的下一个字符的读写指针移到指向 'G' 字符上,然后再用 `fgetc()` 函数读取出 'G' 字符,并用 `printf()` 函数将它显示在屏幕上。

2. 对一个字符串的读写函数

(1) 一个字符串的读函数 `fgets()`

该函数的功能是从指定的文件中读取一个字符串放到指定的字符数组中。该函数的调用格式如下:

```
fgets(s, n, fp)
```

其中, `fgets` 是该函数的函数名,该函数有3个参数: `s` 是一个用来存放读取出来的字符串的字符数组名,或字符指针名; `n` 是用来指定读取字符的个数,其中包含字符串结束 '\0',实际上读取字符为 `n-1` 个; `fp` 是被读取字符串的文件指针。该函数是从被打开的由 `fp` 文件指针所指向的文件中读取 `n-1` 个字符送到所指定的字符数组 `s` 中。在实际操作中,不一定每次都读取 `n-1` 个字符,当读取到换行符或文件结束符时,都将结束读取操作将已读取的字符送到数组 `s` 中。该函数的返回值是指向字符数组 `s` 的指针,出错时返回 `NULL`。读入到 `s` 中的字符自动加 '\0'。

(2) 一个字符串的写函数 `fputs()`

该函数的功能是将指定数组中的字符串写到由文件指针所指向的文件中。该函数的调用格式如下:

```
fputs(s, fp)
```

其中, `fputs` 是该函数的函数名,该函数有2个参数:一个是存放已知字符串的字符数组 `s`,另一个是被打开文件的文件指针 `fp`。该函数是用来将字符数组 `s` 中的字符串写到 `fp` 文件指针指向的文件中。正常时函数返回写入文件的字符个数,出错时返回 `NULL`。参数 `s` 可以是字符数组名、字符指针名,也可以是一个字符串常量。

[例10.12] 分析下列程序的输出结果。注意 `fgets()` 和 `fputs()` 函数的使用方法。

```
#include <stdio.h>
char s[][10] = {"CHINA", "AUSTRALIA", "CANATA"};
int x[] = {6, 10, 7};
main()
{
    int i;
    char a[3][10];
    FILE *fp;
    fp = fopen("abc.txt", "w");
    for(i = 0; i < 3; i++)
        fputs(s[i], fp);
    fclose(fp);
    fp = fopen("abc.txt", "r");
    for(i = 0; i < 3; i++)
        fgets(a[i], x[i], fp);
    for(i = 0; i < 3; i++)
```

```

        printf("%s\n", *(a+i));
    fclose(fp);
}

```

执行该程序输出结果如下所示：

```

CHINA
AUSTRALIA
CANADA

```

说明：该程序首先打开一个文件 abc.txt，然后使用 fputs() 函数将3个字符串写到被打开的文件中，再将该文件关闭。接着，又打开该文件，使用 fgets() 函数从该文件中读取3个字符串并将它存放到字符数组 a 中，再把它们显示在屏幕上，最后关闭文件。

该程序还可以进一步简化成如下形式：

```

#include <stdio.h>
char s[][10]={"CHINA","AUSTRALIA","CANADA"};
int x[]={6,10,7};
main()
{
    int i;
    FILE *fp;
    fp=fopen("abc.txt","w+r");
    for(i=0;i<3;i++)
        fputs(s[i],fp);
    rewind(fp);
    for(i=0;i<3;i++)
    {
        fgets(s[i],x[i],fp);
        printf("%s\n",*(s+i));
    }
    fclose(fp);
}

```

该程序的执行结果与前面程序相同。该程序中出现一个新的函数 rewind(fp)，该函数在本章后面定位函数中要讲到，它的功能是将 fp 所指向的文件的文件读写指针移到文件头。这一函数的使用保证了 fgets() 函数是从 fp 所指向的文件的开头进行读取。

3. 对一个数据块的读写函数

(1) 数据块的读函数 fread()

该函数的功能是从指定的文件中读取一个数据块到指定的内存缓冲区中，数据块的大小取决于数据块中数据项的大小(字节数)和数据项的项数。该函数的调用格式如下：

```
fread(ptr, size, nitem, fp)
```

其中，fread 是该函数的函数名，该函数有4个参数分别是：ptr 是一个指针，用来指出数据块在内存的起始位置；size 表示数据块中每个数据项的大小，以字节为单位；nitem 表示数据块中数据项数；fp 是一个文件指针，用它指出读取数据块的文件。例如，在一台16位机上从某个文件中读取100个浮点数，放到数组 buff 中，可写成如下格式：

```
fread (buff,4, 100,fp);
```

或者,

```
fread (buff,sizeof(float),100,fp);
```

显然,后一种格式也适用于32位机。

该函数正常时返回实际读取的数据项数,非正常时返回0。

(2) 数据块的写函数 fwrite()

该函数的功能是将指定的内存缓冲区中的数据块中所有数据项写到所指定的文件中。所写数据块的大小是由数据块中数据项的大小和项数决定的。该函数的调用格式如下所示:

```
fwrite (ptr,size,nitem,fp);
```

其中,fwrite 是该函数的名字,它有4个参数,每个参数的含意与 fread()函数参数相同。该函数就是将由 ptr 所指向缓冲区内的 nitem 个数据项(每个数据项为 size 个字节)的数据块写到由 fp 所指向的文件中。该函数正常返回实际写入文件的数据项数。

[例10.13] 将一组浮点数写入指定的文件中,并从该文件中读出显示在屏幕上。

程序内容如下:

```
#include <stdio.h>
float f[]={1.2,3.4,5.6,7.8,9.0};
main()
{
    FILE *fp;
    int i;
    float g[10];
    if((fp=fopen("float.dat","w"))==NULL)
    {
        printf("Can't open float.dat.\n");
        exit(1);
    }
    fwrite(f,sizeof(float),5,fp);
    fclose(fp);
    if((fp=fopen("float.dat","rb"))==NULL)
    {
        printf("Can't open float.dat.\n");
        exit(2);
    }
    fread(g,sizeof(float),5,fp);
    for(i=0;i<5;i++)
        printf("%8.2f",g[i]);
    printf("\n");
    fclose(fp);
}
```

执行该程序输出结果如下所示:

```
1.20  3.40  5.60  7.80  9.00
```

说明:该程序中,打开文件 float.dat,使用 fwrite()函数将数组 f 中的5个浮点数写入被打开的文件中,然后关闭该文件。接着,又重新打开 float.dat 文件,再使用 fread()函数从该文件中读取5个浮点数,并存放在数组 g 中,最后通过打印输出 g 数组各元素的值,将5个浮点数显

示在屏幕上。该程序中说明了 `fwrite()` 函数和 `fread()` 函数的使用方法。

4. 一般文件的格式读写函数

(1) 一般文件的格式读函数 `fscanf()`

该函数的功能是按指定的格式从一个指定的文件中读取数据,给该函数参数表中各变量赋值。该函数的调用格式如下:

```
fscanf(fp,"(控制串)",(参数表))
```

其中, `fscanf` 是该函数的文件名,该函数中的参数除了 `fp` 是一个被打开文件的文件指针外,其余参数与 `scanf()` 完全相同。该函数与标准格式读函数的区别仅在于前者是从指定的某个文件中读取数据,而后者是从键盘上读取数据。如果将该函数中的 `fp` 指定为 `stdin`,则它与 `scanf()` 函数是等同的。

(2) 一般文件的格式写函数 `fprintf()`

该函数的功能是按指定的格式将该函数参数表中给定的表达式的值写到指定的文件中。调用格式如下所示:

```
fprintf(fp,"(控制串)",(参数表))
```

其中, `fprintf` 是该函数的名字,该函数的参数中除了 `fp` 是某个文件的指针外,其余参数与标准格式写函数 `printf()` 函数完全相同。`fp` 是该函数要写入数据的文件指针,如果 `fp` 为 `stdout` 则该函数 `printf()` 函数等同。因此,该函数与 `printf()` 函数的区别仅在于前者是将数据写到指定的某个文件中,而后者将数据写到显示屏幕上。例如,已知

```
int a=34;
```

```
float b=56.78;
```

而语句

```
fprintf(fp,"%d,%8.2f",a,b);
```

将变量 `a` 的值按 `%d` 格式,将变量 `b` 的值按 `%8.2f` 格式写到由 `fp` 文件指针所指向的文件中,于是 `fp` 所指向的磁盘文件上便有如下格式的字符串:

```
34,1111156.78
```

还可以使用 `fscanf()` 函数将某个磁盘文件上的上述字符串读出,并赋值给两个变量,使用的函数格式如下:

```
fscanf(fp,"%d,%f",&a,&b);
```

这时,该函数将从 `fp` 文件指针所指向的文件读出 34 赋给变量 `a`,读出 56.78 赋给变量 `b`。

使用上面讲述的两个函数 `fscanf()` 和 `fprintf()` 对磁盘文件读写是很方便的,但是由于这两个函数在输入时要将 ASCII 码转换为二进制形式,在输出时又要将二进制形式转换为 ASCII 码,这样要花费一些转换时间,而使用 `fread()` 和 `fwrite()` 函数就不需要进行这种转换,所以在频繁交换数据的情况下,还是使用 `fread()` 和 `fwrite()` 好些。

[例10.14] 编写一个程序,要求从键盘上输入的一个字符串和一个十进制整数写到一个文件中,然后再将它们从文件中读出,显示在屏幕上。编程中要使用 `fscanf()` 和 `fprintf` 函数。

程序内容如下:

```
#include <stdio.h>
main()
{
```

```

int a;
char s[50];
FILE *fp;
if((fp=fopen("t.txt","w"))==NULL)
{
    puts("can't open file.\n");
    exit(1);
}
printf("Input a and s: ");
fscanf(stdin,"%d%[^\\n",&a,s);
fprintf(fp,"%d\\t%s",a,s);
fclose(fp);
if((fp=fopen("t.txt","r"))==NULL)
{
    puts("can't open file.\n");
    exit(2);
}
fscanf(fp,"%d\\t%[^\\n",&a,s);
fclose(fp);
fprintf(stdout,"%d,%s\\n",a,s);
}

```

执行该程序显示如下信息:

Input s and a: 137, this is a string ↵

输出结果如下:

137, This is a string

说明: 该程序中使用了 `fscanf()` 和 `fprintf()` 函数。先是使用 `fscanf()` 函数从键盘上读取两个数据给变量 `s` 和 `a` 赋值, 这时 `fp` 参数用 `stdin`。然后, 使用 `fprintf()` 函数, 将变量 `s` 和 `a` 中的数据写到 `fp` 所指向的文件 (`t.txt`) 中。最后, 又使用 `fscanf()` 函数从 `fp` 的指向的文件读取数据给变量 `s` 和 `a` 赋值。于是, `s` 和 `a` 将从文件 `t.txt` 中获得一个字符串和一个 `int` 型值, 再用 `fprintf()` 函数将它们显示到屏幕上, 这时 `fp` 参数用 `stdout`。

该程序中还需要说明的一点是在 `fscanf()` 函数的格式符中出现了以前没有见过的 `%[^\\n]` 格式符。该格式仅用于输入字符串, 由一对方括号 `[]` 来指定允许输入的字符, 例如, `[abc]` 表示允许输入的字符是 `a`, `b` 和 `c`, 其他字符不允许输入, 即为非法字符。如果在允许输入的字符集前面加 `^` 符号, 则表示除 `^` 符号后面的符号都允许输入, 例如, `[^abc]` 表示除 `a`, `b` 和 `c` 外的其他字符都可以输入。而这里使用的 `[^\\n]` 是指除了换行符 (`\\n`) 外, 其余字符都是合法的。使用这种格式符可以在输入的字符串中出现空格符, 否则字符串中出现空格符将作为结束符。因此, 该程序中, 为了在输入的字符串中输入空格符而使用了这种格式。

10.3.3 文件定位函数及其使用

文件定位函数包含有定位读写指针函数、归位读写指针函数和返回读写指针函数等。这里的读写指针是指当打开一个文件时系统自动建立一个标识文件中当前字符位置的指针, 该指针随着对文件的读写操作而不断的移动。例如, 为了读一个文件而将该文件打开, 这时读写指针指在文件头, 随着该文件中的字符不断被读出, 读写指针将向文件尾方向移动, 该文件全部

读完,则读写指针指向文件尾。读写指针不同于文件指针,读者要将二者区分开。

1. 定位读写指针函数 fseek()

该函数的功能是将文件的读写指针从某个位置移到指定的位置,该函数将为 C 语言对文件的随机读写提供了方法。该函数调用格式如下:

fseek(fp,〈偏移量〉,〈起始位置〉)

其中,fseek 是该函数的函数名;fp 是指向被操作文件的文件指针;〈偏移量〉是表示移动当前读写指针的距离量,该参数的类型为 long int 型;〈起始位置〉是偏移量的相对位置。例如,起始位置为文件头,偏移量为50,则表示将读写指针移到相对文件头距离为50个字节的位置。起始位置的设置方法有三:

0 表示相对于文件头

1 表示相对于文件的当前位置

2 表示相对于文件尾

实际中,常用宏定义来替代起始位置,规定如下:

SEEK_SET 表示文件头

SEEK_CUR 表示当前位置

SEEK_END 表示文件尾

例如,

fseek(fp,200L,0); 将读写指针移到离文件头200个字节处。

fseek(fp,80L,1); 将读写指针移到离当前位置80个字节处。

fseek(fp,-50L,0); 将读写指针移到从文件尾向后退50个字节处。

该函数一般用于二进制文件。如果用于文本文件要发生字符转换,计算位置时会发生误差。

2. 归位读写指针函数 rewind()

该函数的功能是将某个文件的读写指针归位于文件头。该函数的调用格式如下:

rewind(fp)

其中,rewind 是该函数的函数名,fp 是被操作文件的文件指针。使用该函数后,会使被操作文件的读写指针指向文件头。该函数的功能与下列函数功能相同。

fseek(fp,0L,0);

3. 返回读写指针函数 ftell()

该函数的功能是返回指定文件当前读写指针的位置,该位置是用相对于文件头所相隔的字节数来表示。例如,该函数返回某个文件的当前读写指针的位置是100字节,即表示当前读写指针在离文件头有100个字节处。该函数调用格式如下:

ftell(fp)

它返回一个表示字节数的 long int 型数值。

[例10.15] 建立一个数据文件,随机读取其中的某个数据。

使用 fprintf()函数建立一个数据文件 xy.dat,然后,指定从某个数据起连续读出若干个数据,最后,再读出这组数据的起始数据。

程序内容如下:

```

#include <stdio.h>
main()
{
    int i,x,y;
    FILE *fp;
    fp=fopen("xy.dat","wb+rb");
    for(i=0;i<20;i++)
        fprintf(fp,"%5d",i+1);
    printf("\nInput x: ");
    scanf("%d",&x);
    for(i=0;i<5;i++)
    {
        fseek(fp,(long)(5*(x-1+i)),0);
        fscanf(fp,"%d",&y);
        printf("%d\t",y);
    }
    fseek(fp,(long)(5*x-5),0);
    fscanf(fp,"%d",&x);
    printf("\n%d\n",x);
    fclose(fp);
}

```

执行该程序运行结果如下：

```

Input x:10
10  11  12  13  14
10

```

说明：该程序先打开一个文件 xy.dat，以二进制数的读写方式，用 for 循环方法向文件中写入20个整型数，这里使用的是 fprintf() 函数。再从键盘上键入一个数值赋给 x，表示从文件中第 x 个数据项开始读取数据，并将它显示在屏幕上。本程序中，指定从第10个数据项开始，连续读出5个数据项，暂存在变量 y 中，输出到显示屏幕上，这里使用 fseek() 函数进行定位，让读写指针移到第10个数据项，并从该数据项开始输出。程序中又使用 fseek() 函数重新定位读写指针，使它再指向第10个数据项，再读取该数据，输出显示为10。

[例10.16] 编写一个程序使用 ftell() 函数估算一个文件的大小。

```

#include <stdio.h>
main()
{
    long i;
    FILE *fp;
    if((fp=fopen("ab.txt","r"))==NULL)
    {
        printf("File can't open.\n");
        exit(1);
    }
    fseek(fp,0L,2);
    i=ftell(fp);
}

```



```
printf("file size: %ld\n",i);  
}
```

执行该程序输出结果如下:

```
file size: 20
```

说明: 该程序可用来估算一个文件的长度或大小。本例中, 估算文件 ab.txt 的大小为20个字节, 该文件是本章例10.11中的一个文件。函数 ftell() 是用来返回当前读写指针到该文件头相距的字节数。

10.4 介绍常用的其他函数

这节中介绍一些 C 语言中常用的其他函数。这些函数大多数是被放到指定的 .h 文件中, 使用时应将包含所用函数的 .h 文件。C 语言编译系统所提供的函数很多, 使用时要参阅有关编译系统的说明书, 还应该注意不同编译系统不仅在所提供的函数多少上有区别, 而且有的函数所在的 .h 文件也不同。这里只介绍几种常用的函数, 还有许多函数读者在使用时再去查阅有关资料。例如, conio.h 中的屏幕函数, graphics.h 中的图形函数等, 本书将不再介绍。

10.4.1 动态存储分配函数

动态存储分配函数不同的编译系统所提供的多少不同。有的编译系统放在 malloc.h 中, 有的编译系统放在 stdlib.h 中。常用的有如下几种:

1. 内存分配和释放函数 malloc() 和 free()

这些函数前面已经出现过。malloc() 函数的功能是动态地向内存申请指定大小的空间, 所指定的大小用字节数表示。其调用格式如下:

```
malloc(size)
```

其中, malloc 是该函数名, size 是用来表示所申请的内存大小的字节数。该函数的返回值为 void * 型, 即为某种类型的指针, 使用时将其返回值强制成某种类型的指针。当申请成功时该函数返回所申请内存的缓冲区的首地址, 否则返回 NULL。

例如, 实际中申请50个整型变量的内存空间可以这样使用:

```
int *p;  
p=(int *)malloc(50*sizeof(int));  
if(p==NULL)  
{  
    printf("out memory!\n");  
    exit(1);  
}
```

当申请内存空间时, 总是要判断一下是否申请成功。如果申请成功了, 继续往下进行; 如果申请失败, 则应退出该程序。

free() 函数的功能是用来释放使用 malloc() 函数分配的内存空间。由于机器中用来进行动态分配的内存空间是有限, 及时释放不用的内存空间是十分重要的。该函数调用格式如下:

```
free(p)
```

其中,free 是该函数的名字,p 是要被释放的内存空间的首地址,也是使用 malloc()函数所分配的内存空间的返回值。执行该函数则使指针 p 所指向的内存区域被释放。

2. 内存分配函数和释放函数 calloc()和 cfree()

calloc()函数也是一个用来分配内存地址的函数,使用该函数时表示分配内存区域的大小由两个参数决定,一是数据对象的大小,即占内存的字节数,二是数据对象的多少。该函数的调用格式如下:

```
calloc(n,size)
```

其中,calloc 是该函数名字,n 是申请存放的数据对象的数目,size 是每个存放的数据对象所占内存的字节数。该函数的返回值情况与 malloc()函数相同。该函数与 malloc()函数的区别还在于该函数对分配的内存空间进行初始化,即将所分配的内存空间清为0或空,而 malloc()函数对所分配的内存空间不进行初始化,即所分配的内存空间中还保留着以前的剩余数据。

cfree 函数是用来释放由 calloc()函数所占用的内存空间。其格式如下:

```
cfree(p)
```

其中,p 是一个指向内存中要被释放空间的首地址,即调用 calloc()函数分配空间时的返回值。

3. 改变已分配内存空间大小的函数 realloc()

该函数是用来改变已分配的内存空间的大小的。既可以将原来空间变大,也可以将原来空间变小。该函数格式如下:

```
realloc(p,size)
```

其中,realloc 是该函数的名字,p 是指向已分配的内存空间的指针,即为所要改变大小的内存空间,size 是改变后的内存空间大小。该函数的返回值是指向该内存区域的指针。

10.4.2 系统调用函数

C 语言中提供一个支持系统命令的函数,即系统调用函数 system()。该函数调用格式如下:

```
system("〈系统命令字〉")
```

其中,system 是该函数的函数名,其参数是一个字符串,该串是系统命令字。它可以是系统的各种命令,例如,在 DOS 系统下,可以是:

```
system("date");
```

该函数将直接执行 DOS 系统下的 date 命令,将当前的日期显示在屏幕上。

使用该函数可以将在一个程序中执行另一个编译连接好的可执行文件(即命令),这将为程序的嵌套执行提供了方便。

[例10.17] 使用系统调用函数 system()在一个程序中运行另一个程序。

```
main()
{
    printf("Type l10-16.c:\n");
    system("type l10-16.c");
    printf("Run output:\n");
    system("l10-16");
}
```

执行该程序运行结果如下:

Type l10-16.c:

```
#include <stdio.h>
main()
{
    long i;
    FILE *fp;
    if((fp=fopen("ab.txt","r"))==NULL)
    {
        printf("File can't open. \n");
        exit(1);
    }
    fseek(fp,0L,2);
    i=ftell(fp);
    printf("file size: %ld\n",i);
}
```

Run output:

file size: 20

说明:该程序中两次调用 system() 函数,第一次调用是将例10.16程序内容输出显示在屏幕上,l10-16.c 是例10.16程序文件名,这次函数调用是执行下述 DOS 命令:

type l10-16.c

第二次调用 system() 函数是输出 l10-16.c 文件的结果,l10-16.c 经编译连接后生成可执行文件 l10-16.exe。这次函数调用是执行下述 DOS 命令:

l10-16

于是将 l10-16.c 文件的输出结果显示在屏幕上。

10.4.3 字符函数

C 语言中提供了一些关于字符的函数,这些函数多数是宏定义,它们被放在 ctype.h 文件中,一般情况下,要使用这些函数时,应该先将它们包含在你的程序中,即可使用下述命令:

```
#include <ctype.h>
```

这些函数列表在附录2中。例如,检查一个整型变量 c 是否是字母的函数其格式如下:

```
int isalpha(c)
```

```
int c;
```

该函数返回值是 int 型数。如果返回值为1,说明 c 是字母;如果返回值为0,则说明 c 不是字母。

又例如,将一个小写字母转换为大写字母的函数 toupper(),其格式如下:

```
int toupper(c)
```

```
int c;
```

该函数将小写字母 c 转换为大写字母返回,对非小写字母,则不改变。

还有许多函数见附录2。

10.4.4 常用数学函数

C 语言中提供了许多常用的数学函数,使用它们可进行许多数学运算,例如,求绝对值、求平方根、求对数以及三角函数运算等。它们被存放在头文件 `math.h` 中,在程序中,要调用这些函数时,事先要包含 `math.h` 文件。关于这些数学函数列表在附录3中。下列仅举几个例子说明其用法。例如,求一个数的绝对值的函数 `fabs()`,其格式如下:

```
double fabs(x)
```

```
double x;
```

该函数的功能是求 x 的绝对值,执行该函数返回 x 的绝对值。

又例如,求一个数的平方根函数 `sqrt()`,其定义格式如下:

```
double sqrt(x)
```

```
double x;
```

执行该函数,则返回参数 x 的平方根。

又例如,求某个数的若干次幂函数 `pow()`,其格式如下:

```
double pow(x,y)
```

```
double x,y;
```

执行该函数将返回 x 的 y 次幂,即 x^y 的值。

又例如,求某个角度的正弦值函数 `sin()`,其格式如下:

```
double sin(x)
```

```
double x;
```

执行该函数将返回参数 x (用弧度表示)的正弦值。

其他函数参见附录3。

还有屏幕函数和图形函数请参见有关资料,这些函数都被存放在各自的头文件中,例如,屏幕函数被存放在 `conio.h` 中,图形函数被存放在 `graphics.h` 中,调用这些函数时不要忘记包含存放它们的头文件。

练 习 题

1. C 语言文件有何特点?如何理解“C 语言文件是字符流”?
2. 什么是文件指针?什么是读写指针(又称位置指针)?两者有何区别?
3. 什么是标准文件?什么是一般文件?两者有何区别?
4. 标准文件有哪些读写函数?这些函数的调用格式如何?
5. 标准格式读函数 `scanf()` 中,控制串中格式符与前面的 % 之间允许哪些修饰符?它们的作用是什么?
6. 标准格式写函数 `printf()` 中,控制串中格式符与前面的 % 之间允许哪些修饰符?它们的作用是什么?
7. `scanf()` 函数在使用时应注意些什么?如何清除缓冲区中的剩余字符?
8. 在 `scanf()` 函数的格式符中 %s 和 %c 有何区别?在使用时应该注意些什么?
9. 打开文件函数 `fopen()` 在何时使用?该函数返回值是什么?打开文件的方式有哪些?
10. 一般文件的读写函数有哪些?这些函数的调用格式如何?
11. 文件定位函数有哪些?它们各自的作用是什么?
12. `fseek()` 函数有哪些参数?如何使用该函数来实现对文件的随机读写?

13. 动态存储分配函数有哪些?如何使用这些函数进行动态存储分配?

14. system()函数有何功能?

15. 头文件 ctype.h 和 math.h 中都有哪些常用函数?

作业题

1. 选择填空

(1) C 语言文件的读写()。

- A. 只能用顺序存取方式
- B. 只能用随机存取方式
- C. 可以顺序存取,也可以随机存取
- D. 只能从文件头开始存取

(2) 下列()操作不能将读写指针定位在文件头。

- A. rewind(fp) B. fseek(fp, 0L, 0)
- C. fseek(fp, 0L, 2) D. fopen("文件名", "r")

(3) 使用 fopen() 函数打开一个文件时,此时文件的读写指针()。

- A. 一定在文件头 B. 一定在文件尾
- C. 不确定 D. 可能在文件头,也可能在文件尾

(4) feof() 函数()。

- A. 只能用于文本文件
- B. 只能用于二进制文件
- C. 可用于文本文件和二进制文件
- D. 不可用于文本文件和二进制文件

(5) 在 fgets(str, n, fp) 函数中,下列条件()不可用来结束读入操作。

- A. 换行符 B. EOF
- C. 空格符 D. 已读入 n-1 个字符

(6) 使用 fseek() 函数,将文件的读写指针移到文件结束,则在下列格式的下划线上应填写()。

fseek(fp, 0L, _);

- A. 0 B. 1 C. 2 D. 任意

(7) 关闭文件函数 fclose() 中有一个参数,该参数是()。

- A. 要关闭的文件的读写指针
- B. 要关闭的文件的名字
- C. 要关闭的文件指针
- D. 要关闭的文件的名字和读写指针都可以

(8) 只是为了从一个文件中读取数据,在打开该文件时,打开方式应选择()。

- A. "r+" B. "r" C. "a" D. "rb+"

(9) 使用 feof() 函数可以判断文件是否结束,当文件结束时,该函数的返回值是()。

- A. 1 B. -1 C. 0 D. 任意

(10) 下列语句的功能是()。

fseek(fp, 100L, 1);

- A. 将 fp 所指向的文件的读写指针移到距文件头有100个字节处
- B. 将 fp 所指向的文件的读写指针移到距文件尾有100个字节处

C. 将 fp 所指向的文件的读写指针移到当前读写位置前100个字节处

D. 将 fp 所指向的文件的读写指针移到当前读写位置后100个字节处

2. 判断下列描述正确与否,对者划√,错者划×。

(1) C 语言中读写函数都是由库函数来提供的。

(2) C 语言中对文件的写操作是指把文件中的数据写到内存中去,又称为输出操作。

(3) 在调用 C 语言库函数实现对一般文件的操作时,应包含 stdio.h 文件。

(4) 关闭文件函数 fclose()有一个参数,该参数要求是待关闭文件的名字。

(5) 当使用打开文件函数 fopen()为了写操作打开一个文件时,该文件一定要存在,否则打开操作失败。

(6) 使用"a+"方式打开的文件只能写而不能读。

(7) 使用 fopen()函数打开文件时,文件的读写指针都指向文件头。

(8) 在一个程序中,有 n 个文件,只要用一个文件指针就可以了。

(9) 定位函数 fseek()只能用于二进制文件不能用于文本文件。

(10) 标识文件结束的是文件结束符,每个文件都必须有一个文件结束符。

3. 指出下列程序段中的语法错误。

(1)

```
float x,y;  
scanf("%f",x,y);  
:
```

(2)

```
:  
a=5;  
b=7.52;  
printf("%D,%F\n",a,b);  
:
```

(3)

```
:  
FILE fp,fg;  
fp=fopen("f1.c",r);  
fg=fopen("f2.c",w);  
:  
fclose(fp,fg);  
:
```

(4)

```
:  
long i;  
fp=fopen("f1.c",r);  
while (feof(fp))  
{  
    c=fgetc(fp);  
    putchar(c);  
}  
fclose (fp);  
i = ftell(fp);  
printf("%d\n",i);  
:
```

(5)

```

        :
FILE fp;
if(fp=fopen("f1.c",r)==EOF)
{
    puts("can't open file. \n");
    break;
}
        :

```

4. 分析下列程序的输出结果

(1)

```

#include <stdio.h>
main()
{
    char ch;
    FILE *fp;
    if((fp=fopen("xy.txt","w"))==NULL)
    {
        puts("can't open file. \n");
        exit(1);
    }
    while((ch=getchar())!='#')
        fputc((ch>='a' && ch<='z')?ch-'a'+'A':ch,fp);
    fclose(fp);
    system("type xy.txt");
}

```

当键入如下信息:

```

abcDEF✓
mnPQRS✓
TUVxyz#
输出信息是什么?

```

(2)

```

#include <stdio.h>
main()
{
    FILE *fp;
    char a[][10]={"abcd","efghij","klmno","pqrst",
                  "vwxyz"};
    int i;
    char x;
    fp=fopen("xyz.txt","w+");
    for(i=0;i<5;i++)
        fprintf(fp,"%-10s",a[i]);
    for(i=0;i<5;i++)
    {
        fseek(fp,(long)(i*10L),0);

```

```

        fscanf(fp, "%c", &x);
        printf("\n%c", x);
    }
    fclose(fp);
    putchar('\n');
}

```

(3)

```

#include <stdio.h>
char a[][10]={"FORTRAN","COBOL","PASCAL","Turb C"};
main()
{
    int i;
    char b[4][8];
    FILE *fp;
    fp=fopen("abc.txt","w");
    for(i=0;i<4;i++)
    {
        fputs(a[i],fp);
        fputs("\n",fp);
    }
    fclose(fp);
    fp=fopen("abc.txt","r");
    for(i=0;i<4;i++)
    {
        fgets(b[i],80,fp);
        printf("%s",*(b+i));
    }
    fclose(fp);
}

```

(4)

```

#include <stdio.h>
char pc[][10]={"Wang","professor","computer",
               "Li","professor","chemistry",
               "Ma","assistant","computer",
               "Lu","lecturer","physic"};
int age[]={52,50,27,33};
main()
{
    int i,j;
    char pb[12][10];
    int pd[4];
    FILE *fp;
    fp=fopen("xyz.txt","w");
    for(i=0,j=0;i<12&& j<4;i+=3,j++)
    {
        fprintf(fp,"%8s%10s%10s",pc[i],pc[i+1],pc[i+2]);
    }
}

```



```

        fprintf(fp, "%5d", age[j]);
    }
    fclose(fp);
    fp=fopen("xyz.txt", "r");
    for(i=j=0; i<12&& j<4; i+=3, j++)
    {
        fscanf(fp, "%8s%10s%10s", pb[i], pb[i+1], pb[i+2]);
        fscanf(fp, "%5d", &pd[j]);
    }
    for(i=j=0; i<12&& j<4; i+=3, j++)
    {
        printf("%-8s%-10s%-10s", pb[i], pb[i+1], pb[i+2]);
        printf("%5d\n", pd[j]);
    }
    fclose(fp);
}

```

(5)

```

#include <stdio.h>
main()
{
    int i, x, n=sizeof(float);
    float a[20], b[20];
    FILE *fp;
    printf("Input x: ");
    scanf("%d", &x);
    fp=fopen("xyz.dat", "wb+");
    if(fp==NULL)
    {
        printf("file can't open. \n");
        exit(1);
    }
    for(i=0; i<20; i++)
        a[i]=1.11*(i+1);
    fwrite(a, n, 20, fp);
    rewind(fp);
    fread(b, n, 20, fp);
    for(i=0; i<x; i++)
        printf("%8.2f", b[i]);
    fclose(fp);
    printf("\n");
}

```

5. 编程

(1) 有10个学生, 每个学生有5门课程的成绩, 编程输入学生的数据: 学号, 姓名和各科成绩, 计算出每个学生的总成绩和平均成绩并将以上所有数据放到一磁盘文件 student.txt 中。

(2) 读出上题中 student.txt 文件内的数据, 按学生总成绩由高到低进行排序, 并将已排序的学生姓名和总成绩写到另一个文件 stu_sort.txt 中。

(3) 已知文本文件 f1.txt 的内容为 abc mnp xgz 字符串,文件 f2.txt 的内容为 defghstvw 字符串,编写一程序将 f1.txt 和 f2.txt 的文件内容合并以后进行排序(由小到大),然后存放在 fsort.txt 文件中,同时将其排序后的内容显示在屏幕上。

(4) 建立一个电话簿,每个人包含姓名、性别、年龄和电话号码四项。要求:

- ① 将5个人的4项数据存放在 tele.dat 文件中,要求从键盘输入到文件中。
- ② 追加二个人的信息到 tele.dat 文件中。
- ③ 按姓名查找某人的电话号码。
- ④ 按姓名修改某人的电话号码。

将上述功能用菜单选项的办法予以实现。

附 录

附录1 ASCII 编码表

说明：使用此表查字符的 ASCII 码方法如下：该表可查出字符的 ASCII 码的三种表示形式，即十进制、八进制和十六进制三种形式。例如，查字母 A 的 ASCII 码，在 A 上方101是八进制值，A 下方65是十进制值，A 列的最上行4与 A 行的最右列1组成41为十六进制值。

	0	1	2	3	4	5	6	7
0	000 NUL 0	020 DLE 16	040 SP 32	060 0 48	100 @ 64	120 P 80	140 ' 96	160 p 112
1	001 SOH 1	021 DC1 17	041 ! 33	061 1 49	101 A 65	121 Q 81	141 a 97	161 q 113
2	002 STX 2	022 DC2 18	042 " 34	062 2 50	102 B 66	122 R 82	142 b 98	162 r 114
3	003 ETX 3	023 DC3 19	043 # 35	063 3 51	103 C 67	123 S 83	143 c 99	163 s 115
4	004 EOT 4	024 DC4 20	044 \$ 36	064 4 52	104 D 68	124 T 84	144 d 100	164 t 116
5	005 ENQ 5	025 NAK 21	045 % 37	065 5 53	105 E 69	125 U 85	145 e 101	165 u 117
6	006 ACK 6	026 SYN 22	046 & 38	066 6 54	106 F 70	126 V 86	146 f 102	166 v 118
7	007 BEL 7	027 ETB 23	047 , 39	067 7 55	107 G 71	127 W 87	147 g 103	167 w 119
8	010 BS 8	030 CAN 24	050 (40	070 8 56	110 H 72	130 X 88	150 h 104	170 x 120

续表

	0	1	2	3	4	5	6	7
9	011	031	051	071	111	131	151	171
	HT	EM)	9	I	Y	i	y
	9	25	41	57	73	89	105	121
A	012	032	052	072	112	132	152	172
	LF	SUB	*	:	J	Z	j	z
	10	26	42	58	74	90	106	122
B	013	033	053	073	113	133	153	173
	VT	ESC	+	;	K	[k	{
	11	27	43	59	75	91	107	123
C	014	034	054	074	114	134	154	174
	FF	FS	,	<	L	\	l	
	12	28	44	60	76	92	108	124
D	015	035	055	075	115	135	155	175
	CR	GS	-	=	M]	m	}
	13	29	45	61	77	93	109	125
E	016	036	056	076	116	136	156	176
	SO	RS	.	>	N	^	n	-
	14	30	46	62	78	94	110	126
F	017	037	057	077	117	137	157	177
	SI	US	/	?	O	-	o	DEL
	15	31	47	63	79	95	111	127

附录2 ctype.h 文件中所包含的字符函数

函数名	格 式	功 能	返回值
isalnum()	int isalnum(c) int c;	检查 c 是否是字母(大写或小写)或者是数字	返回值为1,是 返回值为0,不是
isalpha()	int isalpha(c) int c;	检查 c 是否是字母(大写或小写)	返回值为1,是 返回值为0,不是
isascii()	int isascii(c) int c;	检查 c 是否是一个 ASCII 码 (c 在 0 至 0x7f 之间为 ASCII 码)	返回值为1,是 返回值为0,不是
isctrl()	int isctrl(c) int c;	检查 c 是否是控制字符 (控制字符 ASCII 码在 0 和 0x1f 之间)	返回值为1,是 返回值为0,不是
isdigit()	int isdigit(c) int c;	检查 c 是否是数字 (0~9)	返回值为1,是 返回值为0,不是
isgraph()	int isgraph(c) int c;	检查 c 是否是可打印字符 (可打印字符 ASCII 在 0x21 至 0x7e 间)	返回值为1,是 返回值为0,不是
islower()	int islower(c) int c;	检查 c 是否是字母 (a~z)	返回值为1,是 返回值为0,不是
isprint()	int isprint(c) int c;	检查 c 是否是可打印字符(包括空格), 其 ASCII 码在 0x21 到 0x7e 间	返回值为1,是 返回值为0,不是
ispunct()	int ispunct(c) int c;	检查 c 是否是标点符号,即除字母 数字和空格以外的所有可打印字符	返回值为1,是 返回值为0,不是
isspace()	int isspace(c) int c;	检查 c 是否是空白符(空格符、水平 制表符和换行符)	返回值为1,是 返回值为0,不是
isupper()	int isupper(c) int c;	检查 c 是否是大写字母(A~Z)	返回值为1,是 返回值为0,不是
isxdigit()	int isxdigit(c) int c;	检查 c 是否是一个 16 进制的字符 (即 0~9, 或 a~f, 或 A~F)	返回值为1,是 返回值为0,不是
tolower()	int tolower(c) int c;	将大写字母 C 转换为小写字母, 对非大写字母,则不改变	C 是大写字母,返回小写 C 是非大写字母,返回不变
toupper()	int toupper(c) int c;	将小写字母 c 转换为大写字母, 对非小写字母,则不改变	c 是小写字母,返回大写, c 是非小写字母,返回不变

附录3 math.h 文件中所包含的数学函数

函数名	格 式	功 能	返回值
acos()	double acos(arg) double arg;	计算 $\cos^{-1}(\arg)$, 即求反余弦值 arg 在 -1 到 1 之间	计算结果
asin()	double asin(arg) double arg;	计算 $\sin^{-1}(\arg)$, 即求反正弦值 arg 在 -1 到 1 之间	计算结果
atan()	double atan(arg) double arg;	计算 $\tan^{-1}(\arg)$ 即求反正切值 arg 在 -1 到 1 之间	计算结果
atan2()	double atan2(x,y) double x,y	计算 $\tan^{-1}(x/y)$	计算结果
cos()	double cos(arg) double arg;	计算 $\cos(\arg)$, 即求余弦值 arg 用弧度表示	计算结果
cosh()	double cosh(arg) double arg;	计算 arg 的双曲余弦值 arg 用弧度表示	计算结果
exp()	double exp(arg) double arg;	计算 e^{\arg} 的值, 即自然数为底的 的指数值	计算结果
fabs()	double fabs(num) double num;	求 num 的绝对值	计算结果
floor()	double floor(num) double num;	求出不大于 num 的最大数	返回该整数的 双精度实数
fmod()	double fmod(x,y) double x,y;	求整除 x/y 的余数	返回余数的 双精度值
frexp()	double frexp(num,exp) double num; int *exp;	将 num 分为数字部分(尾数)x 和以 2 为底的指数部分 n, 即使 $\text{num} = x * 2^n$, 指数 n 存放在 exp 指向的变量中, 返回 x	返回数字部分 x。 $0.5 \leq x < 1$
log()	double log(num) double num;	计算 $\log_e(\text{num})$, 即 $\ln(\text{num})$, 以 e 为底的对数值	计算结果
log10()	double log10(num) double num;	计算 $\log_{10}(\text{num})$, 即以 10 为底的对数 值	计算结果
modf()	double modf(num,p) double num; int *p;	将 num 分为整数部分和小数部分, 将 其整数部分存在指针 p 所指向的变量中, 返回其小数部分	返回 num 被 分解后的小数 部分
pow()	double pow(x,y) double x,y;	计算 x^y 的值	计算结果
sin()	double sin(arg) double arg;	计算 $\sin(\arg)$ 的值, 即求正弦 值, arg 用弧度表示	计算结果
sinh()	double sinh(arg) double arg;	计算 arg 的双曲正弦值	计算结果
sqrt()	double sqrt(num) double num;	计算 $\sqrt{\text{num}}$ 值, 即求平方根 $\text{num} \geq 0$	计算结果
tan()	double tan(arg) double arg;	计算 $\tan(\arg)$ 的值, 即求正切值, arg 用弧度表示	计算结果
tanh()	double tanh(arg) double arg;	计算 arg 的双曲正切函数值	计算结果